Rule-based Program Specialization to Optimize Gradually Typed Code

Francisco Ortin[a,*], Miguel Garcia[a], Seán McSweeney[b]

[a] *University of Oviedo, Computer Science Department, Calvo Sotelo s/n, 33007, Oviedo, Spain*
[b] *Cork Institute of Technology, Department of Computer Science, Rossa Avenue, Bishopstown, Cork, Ireland*

Abstract

Both static and dynamic typing provide different benefits to the programmer. Statically typed languages support earlier type error detection and more opportunities for compiler optimizations. Dynamically typed languages facilitate the development of runtime adaptable applications and rapid prototyping. Since both approaches provide benefits, gradually typed languages support both typing approaches in the very same programming language. Gradual typing has been an active research field in the last years, turning out to be a strong influence on commercial languages. However, one important drawback of gradual typing is the runtime performance cost of the additional type checks performed at runtime.

In this article, we propose a rule-based program specialization mechanism to provide significant performance optimizations of gradually typed code. Our system gathers dynamic type information of the application by simulating its execution. That type information is used to optimize the generated code, reducing the number of type checks performed at runtime. Moreover, program specialization allows the early detection of compile-time type errors, providing static type safety. To ensure the correctness of the proposed approach, we prove its soundness and efficiency properties. The specialization system has been implemented as part of a full-fledged programming language, measuring the runtime performance gain. The generated code performs significantly better than the state-of-the-art techniques to optimize dynamically typed code. Unlike the existing approaches, our system does not consume additional memory resources at runtime, because program specialization is performed statically. Program specialization involves an average compilation time increase from 2% to 11.75%.

Keywords: Gradual typing, program specialization, rule-based systems, type safety, runtime performance

1. Introduction

In the last years, gradual typing has been an active research topic in the area of programming language design, theoretical computer science, and software development [1]. Gradual typing is based on allowing programmers to combine the benefits of static and dynamic typing in the very same programming language [2]. Static typing commonly provides earlier type error detection and more opportunities for compiler optimizations, whereas dynamic typing facilitates the creation of runtime adaptable programs and rapid prototyping [3].

Research in gradual typing has influenced the design and implementation of commercial programming languages. There exist gradually typed languages, such as Visual Basic, Objective-C, Dylan, Boo, Fantom and Cobra, aimed at providing the benefits of both approaches. Some dynamically typed languages, such as Groovy and PHP, have become gradually typed, performing static type checking when the programmer writes explicit type annotations [4, 5]. Additionally, the statically typed C# language has included the `dynamic` type in its version 4.0 [6], indicating the compiler to postpone type checks until runtime.

Gradual typing was first described for Siek and Taha in 2006 [7]. They formally described a functional language combining (static) type annotations and dynamically typed variables. Using different rule-based deductive systems, they specify the syntax, type system and semantics of the proposed gradually typed language, proving that the language is sound[1] (but not *statically* type safe) [7]. Namely, if evaluation terminates, the result is either a value of the expected type or a cast error, but never a type error. This is achieved by adding additional casts to dynamically typed code [2].

Gradually typed languages are not *statically* type safe [2]. This means that dynamically typed variables are type checked at runtime to provide soundness (i.e., to avoid type errors at runtime), because the compiler has not enough information to perform those type checks statically [9]. Unfortunately, these additional runtime type checks cause significant runtime performance penalties [10, 11].

Due to the runtime performance cost of gradual typing, there have been different works aimed at optimizing this kind of languages. Among others, these works include different strategies for compilation [12], tracing Just-In-Time compilation [13], and different techniques to optimize the language runtime [14]. However, the optimization of these languages is still an open issue, since the runtime performance obtained is still below the expected one [10, 12].

[*]Corresponding author

Email addresses: `ortin@uniovi.es` (Francisco Ortin), `garciarmiguel@uniovi.es` (Miguel Garcia), `Sean.McSweeney@cit.ie` (Seán McSweeney)

URL: `http://www.di.uniovi.es/~ortin` (Francisco Ortin),
`http://www.reflection.uniovi.es/miguel` (Miguel Garcia)

[1]Instead of sound, they refer to this property as type safety. However, another property discussed is called statically type safety. Since type safety and soundness are commonly used as synonyms [8], we use soundness to differentiate it from statically type safety.

In this article, we present a formal rule-based system and its implementation in a full-fledged object-oriented language to optimize gradual typing. The main idea is to statically gather type information from dynamically typed code and use it for two purposes: detecting more type errors of dynamically typed code at compile time; and providing better runtime performance by reducing type-checking operations at runtime.

For that purpose, we define a system that simulates the execution of the application as a kind of abstract interpretation aimed at inferring type information of dynamically typed code. The type information inferred is used to specialize the source program into another one where type annotations are added to the dynamically typed variables. The specialized program allows detecting more type errors at compile time, and it is statically type safe when compiler warnings are considered as errors. Moreover, its execution is significantly faster because the specialized program requires fewer type checking operations at runtime.

The main contribution of this work is a rule-based system to specialize gradually typed programs into semantically equivalent code that provides early type error detection and better runtime performance. After proving the soundness of the proposed system, we implement it as part of a full-fledged programming language, measuring the runtime performance gained and comparing it with the state-of-the-art optimization techniques.

The rest of this paper is structured as follows. The next section presents a motivating example to show how the program specializer works. Section 3 formalizes the source gradually typed object-oriented language and the target language used as the output of the specializer. Program specialization is described in Section 4 together with the proofs of its properties. Section 5 describes how the language has been compiled into the .NET platform. An evaluation of runtime performance, memory consumption and compilation time is presented in Section 6. Section 7 depicts the related work and the conclusions and future work are presented in Section 8. We also provide some additional appendices with supporting specifications and proofs.

2. A motivating example

Figure 1 shows an example C# program that combines dynamically and statically typed code. In C#, the `dynamic` type tells the compiler to postpone all the type checking operations for that variable until runtime [6].

The `DistanceToOrigin` method in Figure 1 receives a `dynamic` parameter, so any expression can be passed as an argument. Both `Circumference` and `Rectangle` objects can be safely passed to `DistanceToOrigin` because both types provide appropriate `GetX` and `GetY` methods. This property is called *duck* typing in the dynamic language community: the suitability to perform an operation with an object is determined by the object itself, rather than by its static type [15]. In our example, circumferences and rectangles do not need to share a common type (e.g., `Figure`) to be safely passed to `DistanceToOrigin`. At runtime, it is checked that the `figure` parameter provides the `GetX` and `GetY` methods.

Therefore, the two first invocations to `DistanceToOrigin` in Figure 1 perform the expected computation (computing the distance to the origin), regardless of the dynamic value of `condition`. C# and Visual Basic use reflection[2] to retrieve and invoke the `GetX` and `GetY` methods at runtime, entailing a significant runtime performance cost [16]. In contrast, our proposed system specializes the `DistanceToOrigin` method at compilation time. For the first invocation, a new method version receiving one `Circumference` is created. This new method performs no runtime type checking at all. For the second invocation, another version is produced. In this case, the code only checks if the parameter is `Circumference` or `Rectangle` (there is no other possible option), performing the subsequent cast. We have evaluated this code to be much faster than the use of reflection [17].

The third invocation to `DistanceToOrigin` may produce a runtime type error. Depending on the dynamic value of `condition`, the argument could be a `Triangle`, which does not implement the `GetX` and `GetY` methods. For this reason, the specialized method checks the dynamic type of the argument. One benefit of our system compared to gradual typing languages (e.g., C#) is that it shows a warning message at compilation time (Section 4), stating that `Triangle` does not provide `GetX` and `GetY` methods. Moreover, it generates more efficient code because a single cast is used instead of reflection (Section 5.2).

The fourth invocation to `DistanceToOrigin` passes a `Triangle` as an argument. C# (and any gradually typed language) shows no error at compile time, throwing a `RuntimeBinder` exception at runtime. Since we perform an abstract interpretation of the program, we know statically that the argument type is not valid, so we show an error at compilation time (Section 4), increasing the static type safety of the language.

3. Language design

We describe the input gradually typed formal language FC#$^{\mathrm{G}}$, the intermediate statically typed language used as the output of our program specializer FC#$^{\mathrm{S}}$, the program specialization system (Section 4), and the code generated for the .NET platform (Section 5). Throughout the article, we use the $^{\mathrm{G}}$ superscript to refer to the particular elements of the gradually typed language (FC#$^{\mathrm{G}}$), the $^{\mathrm{S}}$ superscript for the statically typed one (FC#$^{\mathrm{S}}$), and no superscript for the elements common to both languages (e.g., expressions, which follow the same syntax in both languages).

3.1. FC#^{G}

We formalize the FC#$^{\mathrm{G}}$ syntax, type system and semantics. Then, we prove some basic properties and discuss why, unlike in [2] and [18], FC#$^{\mathrm{G}}$ is not type safe.

[2]Although C# uses reflection, once the dynamic type is known at runtime, it is stored in a type cache provided by the Dynamic Language Runtime (DLR).

```
public class Circumference {
  dynamic x, y, radius;
  public dynamic GetX() { return this.x; }
  public dynamic GetY() { return this.y; }
  public Circumference(dynamic x, dynamic y,
                       dynamic radius) {
    this.x = x; this.y = y; this.radius = radius;
} }
public class Rectangle {
  dynamic x, y, width, height;
  public dynamic GetX() { return this.x; }
  public dynamic GetY() { return this.y; }
  public Rectangle(dynamic x, dynamic y,
                   dynamic width, dynamic height) {
    this.x = x; this.y = y;
    this.width = width; this.height = height;
  }
}
class Triangle {
  dynamic x1, y1, x2, y2, x3, y3;
  public Triangle(dynamic x1, dynamic y1,
                  dynamic x2, dynamic y2,
                  dynamic x3, dynamic y3) {
    this.x1 = x1; this.y1 = y1; …
} }
class Node {
  public dynamic info;
  public Node next;
  public Node(dynamic info, Node next) {
    this.info = info; this.next = next;
} }
```

```
class Distance {
  static double DistanceToOrigin(dynamic figure) {
    return Math.Sqrt(Math.Pow(figure.GetX(), 2) +
           Math.Pow(figure.GetY(), 2));
  }
  static dynamic ClosestToOrigin(Node list) {
    if (list == null) return null;
    dynamic element = list.info;
    double min = DistanceToOrigin(element);
    while (list.next != null) {
      list = list.next;
      double distance = DistanceToOrigin(list.info);
      if (distance < min) {
        element = list.info;
        min = distance;
      }
    }
    return element;
  }
  static void Main() {
    Circumference cir = new Circumference(0,0,10);
    Rectangle rec = new Rectangle(1.2,3.7,1.0,2.0);
    Triangle tri = new Triangle(1, 2, -1, 3, 7, 2);
    DistanceToOrigin(cir);
    bool condition = new Random().NextDouble() >= 0.5;
    DistanceToOrigin(condition ? cir : rec);
    DistanceToOrigin(condition ? rec : tri);
    DistanceToOrigin(tri);
    Node list = new Node(rec, new Node(cir, null));
    dynamic fig = ClosestToOrigin(list);
} }
```

Figure 1: A motivating example in C# with statically and dynamically typed code.

3.1.1. FC#^{G} abstract syntax

This language is based on the well-known Featherweight Java (FJ) language, the minimal Java core [19]. FJ omits many features of the Java programming language (interfaces, overloading, assignment, abstract methods and exceptions) to provide a small calculus that models the syntax, semantics and type system of the minimal Java core. Its simplicity allows the rigorous demonstration of key properties such as type safety. FJ embodies many of the central features of the Java programming language, such as class definition, object creation, field access, method invocation, method overriding, method recursion through `this`, subtyping and casting [19]. FJ has been used to formally specify how to include different features to object-oriented languages, including confined types [20], feature-oriented-programming [21], union types [22], approximate data types [23], session types [24] and generics [19].

In this work, we add to FJ the `dynamic` type and modify its syntax to be closer to C# (the language we selected to implement the proposed system). We named the language FC#^{G} , Featherweight C# with gradual typing. FC#^{G} omits many C# features that do not interact with gradual typing in significant ways [2].

The abstract syntax of FC#^{G} is shown with the production rules depicted in Figure 2. A program (P^G) is a sequence of class definitions (CD^G) followed by one expression (e) corresponding to the body of the `Main` method in C#. The meta-variables C range over class names; m ranges over method names and f over field names; and x ranges over variables. As in FJ, $\overline{e}$ is shorthand for a possibly empty sequence $e_1 \ldots e_n$ (and similarly

$$
\begin{array}{llll}
P^G & \in & \text{Program} & ::= \overline{CD^G}\ e \\
CD^G & \in & \text{Class definition} & ::= \texttt{class}\ C\ \texttt{:}\ C\ \texttt{\{}\ \overline{T^G f}\texttt{;}\ K^G\ \overline{M^G}\ \texttt{\}} \\
K^G & \in & \text{Constructor} & ::= C\texttt{(}\overline{T^G f}\texttt{):base(}\overline{f}\texttt{)}\ \texttt{\{}\ \overline{\texttt{this.}f\texttt{=}f\texttt{;}}\ \texttt{\}} \\
M^G & \in & \text{Method} & ::= T^G\, m(\overline{T^G x})\ \texttt{\{ return}\ e\texttt{; \}} \\
e & \in & \text{Expression} & ::= x \mid \texttt{this} \mid e.f \mid e.m(\overline{e}) \mid \texttt{new}\ C(\overline{e}) \\
T^G & \in & \text{Type} & ::= \texttt{dynamic} \mid C
\end{array}
$$

Figure 2: Syntax of FC#$^{\text{G}}$.

for $\overline{CD^G}$, $\overline{f}$, $\overline{M^G}$, etc.). We abbreviate operations on pairs of sequences in the obvious way, writing $\overline{C f}$ for $C_1 f_1, \ldots, C_n f_n$, where n is the length of $\overline{C f}$. We write the empty sequence as $\varnothing$ and denote concatenation of sequences using a comma. The length of a sequence $\overline{x}$ is written $\#(\overline{x})$.

The class declaration `class` C `:` C_{super} `{` $\overline{T^G f}$`;` K^G $\overline{M^G}$ `}` defines a new C class, derived from C_{super}, with $\overline{T^G f}$ fields, the K^G constructor, and the sequence of $\overline{M^G}$ method definitions. For the sake of simplicity, the base class is always specified, even when it is `Object`. `Object` is a distinguished class name whose definition does not appear in the programs.

Each class must define one constructor just aimed at initializing its fields, after calling the base constructor. Methods are defined by specifying the return type, its unique identifier, its parameters, and a single expression representing its method body. Expressions may be variables (x), the `this` keyword, field access, method invocation, and object construction. A type in FC#$^{\text{G}}$ may be a class or the `dynamic` type.

3.1.2. FC#^{G} type system

For the formal description of the type system and the language semantics, we use inference rules (e.g., Figure 3), commonly used in natural deduction logical systems. Each inference rule means: if the statements in the premises listed above the line are established, then the conclusion below the line may be derived. Rules with no premises are axioms (e.g., S-Ref$^{\text{G}}$ in Figure 3).

Figure 3 depicts the subtyping rules that are later used to describe the type system. The subtyping judgment $\overline{CD^G} \vdash C_1 \leq C_2$ means that, in a program with $\overline{CD^G}$ class definitions, C_1 is a subtype of C_2 (any term of type C_1 can safely be used in a context where a term of type C_2 is expected). As in FJ, subtyping is defined by the inheritance relation (S-Super$^{\text{G}}$): a derived class (C_{sub}) is a subtype of its superclass (C_{super}). The subtyping relation is reflexive (S-Ref$^{\text{G}}$) and transitive (S-Trans$^{\text{G}}$).

As described by Siek and Taha, gradual typing is based on the *consistency* relation described in [7]. Later, they combined consistency with subtyping, defining the consistent subtyping relation ($\lesssim$) [2], which is the actual subtyping relation used in the existing implementations of object-oriented gradually typed languages. It is based on the idea that `dynamic` promotes to (is subtype of) any other type (CS-Left$^{\text{G}}$), and the other way round (CS-Right$^{\text{G}}$). The $\lesssim$ relation includes subtyping (CS-Sub$^{\text{G}}$). However, unlike subtyping, the consistent subtyping relation is *not* transitive. So, if $\overline{CD^G} \vdash$

(S-Ref$^{\mathrm{G}}$)

$$\frac{}{\overline{CD^G} \vdash C \leq C}$$

(S-Trans$^{\mathrm{G}}$)

$$\frac{\overline{CD^G} \vdash C_1 \leq C_2 \qquad \overline{CD^G} \vdash C_2 \leq C_3}{\overline{CD^G} \vdash C_1 \leq C_3}$$

(S-Super$^{\mathrm{G}}$)

$$\frac{\overline{CD^G} = CD_1^G \ldots \texttt{class}\, C_{sub}\texttt{:}C_{super}\,\texttt{\{}\overline{T_f^G\,f}\,\texttt{;}\; K^G\,\overline{M^G}\texttt{\}}\;\ldots CD_n^G}{\overline{CD^G} \vdash C_{sub} \leq C_{super}}$$

(CS-Left$^{\mathrm{G}}$)

$$\frac{}{\overline{CD^G} \vdash \mathit{dynamic} \lesssim T^G}$$

(CS-Right$^{\mathrm{G}}$)

$$\frac{}{\overline{CD^G} \vdash T^G \lesssim \mathit{dynamic}}$$

(CS-Sub$^{\mathrm{G}}$)

$$\frac{\overline{CD^G} \vdash C_1 \leq C_2}{\overline{CD^G} \vdash C_1 \lesssim C_2}$$

Figure 3: Subtyping ($\leq$) and consistent subtyping ($\lesssim$).

$C_1 \lesssim \mathit{dynamic}$ and $\overline{CD^G} \vdash \mathit{dynamic} \lesssim C_2$, it is not true that $\overline{CD^G} \vdash C_1 \lesssim C_2$ unless $\overline{CD^G} \vdash C_1 \leq C_2$.

Figure 4 depicts the inference rules for the FC#$^{\mathrm{G}}$ type system. The environment or typing context Γ represents a map associating to each local variable its type in the current scope. The judgment $\overline{CD^G}, \Gamma \vdash e :^G T^G$ means that the expression e has the type T^G in a program with $\overline{CD^G}$ class definitions and the environment Γ. As in FJ, the types of variables (T-Var$^{\mathrm{G}}$) and `this` (T-This$^{\mathrm{G}}$) are taken from the environment. In a method body, Γ maps `this` to the class where the method is defined, and the rest of variables in Γ are the method parameters (see the formalization in W-Class$^{\mathrm{G}}$, Appendix B).

For field access, FC#$^{\mathrm{G}}$ considers two different scenarios. If the type of the object is a class (T-FieldC$^{\mathrm{G}}$), the field type is taken from its class definition, considering inheritance —the auxiliary *fields*, *type* and *method* functions used in Figure 4 are detailed in Appendix A. However, if the object is `dynamic` (T-FieldD$^{\mathrm{G}}$), type checking will be performed at runtime (next section), so the type system just sets `dynamic` to the field type.

For method invocation, if the implicit object (the object used to invoke the method) is not `dynamic` (T-InvC$^{\mathrm{G}}$), the types of the arguments must be consistent subtypes (not just subtypes, as in FJ, to consider `dynamic`) of the types of the parameters, and the type of the invocation is the return type of the method. Similar to field access, if the implicit object is `dynamic` (T-InvD$^{\mathrm{G}}$), no static type checking is done, and the invocation type will also be `dynamic`.

T-New$^{\mathrm{G}}$ checks that the arguments passed to the constructor are consistent subtypes of the parameters (i.e., `dynamic` is considered). This condition is checked recursively in the mandatory invocation of inherited constructors ($\texttt{base(}\overline{f_1}\texttt{)}$).

The additional well-formedness rules required to type check FC#$^{\mathrm{G}}$ are detailed in Appendix B. A well-formed program $P^G = \overline{CD^G}\, e$ will be accepted by the type

(T-Var$^{\mathrm{G}}$)

$$\frac{}{\overline{CD^G}, \Gamma \vdash x :^G \Gamma(x)}$$

(T-This$^{\mathrm{G}}$)

$$\frac{}{\overline{CD^G}, \Gamma \vdash \texttt{this} :^G \Gamma(\texttt{this})}$$

(T-FieldC$^{\mathrm{G}}$)

$$\frac{\overline{CD^G}, \Gamma \vdash e :^G C \qquad fields(\overline{CD^G}, C) = T_1^G f_1 \dots T^G f \dots T_n^G f_n}{\overline{CD^G}, \Gamma \vdash e.f :^G T^G}$$

(T-FieldD$^{\mathrm{G}}$)

$$\frac{\overline{CD^G}, \Gamma \vdash e :^G dynamic}{\overline{CD^G}, \Gamma \vdash e.f :^G dynamic}$$

(T-InvC$^{\mathrm{G}}$)

$$\frac{\begin{array}{c}\overline{CD^G}, \Gamma \vdash e_{obj} :^G C_{obj} \\ type(method(\overline{CD^G}, C_{obj}, m)) = \overline{T_p^G} \to T_r^G \\ \overline{CD^G}, \Gamma \vdash \overline{e_{arg}} :^G \overline{T_{arg}^G} \\ \overline{CD^G} \vdash \overline{T_{arg}^G} \lesssim \overline{T_p^G}\end{array}}{\overline{CD^G}, \Gamma \vdash e_{obj}.m(\overline{e_{arg}}) :^G T_r^G}$$

(T-InvD$^{\mathrm{G}}$)

$$\frac{\begin{array}{c}\overline{CD^G}, \Gamma \vdash e_{obj} :^G dynamic \\ \overline{CD^G}, \Gamma \vdash \overline{e_{arg}} :^G \overline{T_{arg}^G}\end{array}}{\overline{CD^G}, \Gamma \vdash e_{obj}.m(\overline{e_{arg}}) :^G dynamic}$$

(T-New$^{\mathrm{G}}$)

$$\frac{\begin{array}{c}\overline{CD^G} = CD_1^G \dots \texttt{class}\, C : C_{super}\, \texttt{\{}\overline{T_{f2}^G\, f_2}\texttt{;}\ K^G\ \overline{M^G}\texttt{\}} \dots CD_n^G \\ K^G = C(\overline{T_{p1}^G\, f_1}, \overline{T_{p2}^G\, f_2}) : \texttt{base(}\overline{f_1}\texttt{)}\ \texttt{\{}\overline{\texttt{this.}f_2 = f_2\texttt{;}}\texttt{\}} \qquad \overline{CD^G}, \Gamma \vdash \overline{e_{arg1}} :^G \overline{T_{arg1}^G} \\ \overline{CD^G} \vdash \overline{T_{arg1}^G} \lesssim \overline{T_{p1}^G} \qquad \overline{CD^G}, \Gamma \vdash \overline{e_{arg2}} :^G \overline{T_{arg2}^G} \qquad \overline{CD^G} \vdash \overline{T_{arg2}^G} \lesssim \overline{T_{p2}^G}\end{array}}{\overline{CD^G}, \Gamma \vdash \texttt{new}\, C(\overline{e_{arg1}}, \overline{e_{arg2}}) :^G C}$$

Figure 4: Type system of FC#$^{\mathrm{G}}$.

system when $\overline{CD^G}, \varnothing \vdash e :^G \diamond$ is derived from the existing rules.

3.1.3. FC#^{G} semantics

Figure 5 shows the reduction rules describing the dynamic semantics of FC#$^{\mathrm{G}}$. The judgment $\overline{CD^G} \vdash e_1 \longrightarrow^G e_2$ means that the expression e_1 is evaluated (reduced to) e_2 in one step. We write $\longrightarrow^*$ for the reflexive and transitive closure of $\longrightarrow$.

An expression cannot be reduced any further when it is an *object* (also called *value*). The way we represent an object is with an invocation to its class constructor, where parameters are also objects. Therefore, objects (values) are recursively defined as:

$\sigma \in$ Values $\quad ::= \quad \texttt{new}\ C(\overline{\sigma})$

Unlike FJ, the reduction relation for FC#$^{\mathrm{G}}$ is deterministic. The obvious congruence rules are defined in Appendix C.

R-Field$^{\mathrm{G}}$ evaluates the field access expression, by returning the corresponding expression associated to the requested field (σ_i). Unlike FJ, we must check at runtime whether the f field is actually provided by the object (first premise in the rule), since the addition of the `dynamic` type allows this scenario (i.e., the language is not statically type safe). In case the field is not provided ($f \notin \overline{f}$), R-FieldE$^{\mathrm{G}}$ evaluates the expression to a runtime error holding a descriptive message.

(R-FIELD$^{\text{G}}$)

$$\frac{\mathit{fields}(\overline{CD^G}, C) = T_1^G f_1 \ldots T_i^G f \ldots T_n^G f_n \qquad \overline{\sigma} = \sigma_1 \ldots \sigma_i \ldots \sigma_n}{\overline{CD^G} \vdash \texttt{new}\, C(\overline{\sigma}).f \longrightarrow^G \sigma_i}$$

(R-FIELDE$^{\text{G}}$)

$$\frac{\mathit{fields}(\overline{CD^G}, C) = \overline{T^G f} \qquad f \notin \overline{f}}{\overline{CD^G} \vdash \texttt{new}\, C(\overline{\sigma}).f \longrightarrow^G \mathit{error}(\text{“Field } f \text{ not found”})}$$

(R-INV$^{\text{G}}$)

$$\frac{\mathit{method}(\overline{CD^G}, C_{obj}, m) = T_r^G\, m(\overline{T_p^G\, x})\, \texttt{\{return}\, e_{body}\, \texttt{;\}} \qquad \overline{CD^G} \vdash \overline{C_{arg}} \lesssim \overline{T_p}}{\begin{array}{l}\overline{CD^G} \vdash \texttt{new}\ C_{obj}(\overline{\sigma_{obj}}).m(\texttt{new}\ C_{arg_1}(\overline{\sigma_{arg_1}}), \ldots, \texttt{new}\ C_{arg_n}(\overline{\sigma_{arg_n}})) \longrightarrow^G \\ \quad [\texttt{new}\ C_{arg_1}(\overline{\sigma_{arg_1}})/x_1, \ldots, \texttt{new}\ C_{arg_n}(\overline{\sigma_{arg_n}})/x_n, \texttt{new}\ C_{obj}(\overline{\sigma_{obj}})/\texttt{this}]e_{body}\end{array}}$$

(R-INVE$^{\text{G}}$)

$$\frac{\mathit{method}(\overline{CD^G}, C, m) = \varnothing}{\overline{CD^G} \vdash \texttt{new}\, C(\overline{\sigma_{obj}}).m(\overline{\sigma_{arg}}) \longrightarrow^G \mathit{error}(\text{“Method } m \text{ not found”})}$$

(R-PARE$^{\text{G}}$)

$$\frac{\mathit{method}(\overline{CD^G}, C, m) = T_r^G\, m(\overline{T_p^G\, x})\, \texttt{\{return}\, e\texttt{;\}} \qquad \#(\overline{\sigma_{arg}}) \neq \#(\overline{x})}{\overline{CD^G} \vdash \texttt{new}\, C(\overline{\sigma_{obj}}).m(\overline{\sigma_{arg}}) \longrightarrow^G \mathit{error}(\text{“Wrong number of arguments”})}$$

(R-ARGE$^{\text{G}}$)

$$\frac{\begin{array}{c}\mathit{method}(\overline{CD^G}, C, m) = T_r^G\, m(\overline{T_p^G\, x})\, \texttt{\{return}\, e\texttt{;\}} \\ \#(\overline{x}) = n \qquad \exists\, i \in [1, n]\, .\, \overline{CD^G} \vdash \mathit{not}(C_{arg_i} \lesssim T_{p_i})\end{array}}{\begin{array}{c}\overline{CD^G} \vdash \texttt{new}\ C_{obj}(\overline{\sigma_{obj}}).m(\texttt{new}\ C_{arg_1}(\overline{\sigma_{arg_1}}), \ldots, \texttt{new}\ C_{arg_n}(\overline{\sigma_{arg_n}})) \longrightarrow^G \\ \mathit{error}(\text{“Wrong type of the } i^{th}\text{argument”})\end{array}}$$

Figure 5: Semantics of FC#$^{\text{G}}$.

The same occurs with method invocation. R-INV$^{\text{G}}$ evaluates a method invocation to the method body (σ_{body}), substituting the argument values for the parameters and the implicit object for `this` ($[\sigma/x]e$ represents the substitution of the x variable by the σ value in the expression e). If the implicit object does not provide the method (the only premise in R-INVE$^{\text{G}}$), or the number of parameters is different to the number of arguments ($\#(\overline{\sigma_{arg}}) \neq \#(\overline{x})$ in R-PARE$^{\text{G}}$), or the type of the arguments are not consistent subtypes of the types of the parameters ($\mathit{not}(C_{arg_i} \lesssim T_{p_i})$ in R-ARGE$^{\text{G}}$), a runtime error is produced.

3.1.4. Properties of FC#^{G}

Property 1. *Static type safety of fully annotated FC#^{G} programs.*

If $P^G = \overline{CD^G}\, e_1$ is fully annotated with types and $\overline{CD^G}, \varnothing \vdash e_1 :^G T_1^G$ and

$\overline{CD^G} \vdash e_1 \longrightarrow^{G*} e_2$, then $\overline{CD^G}, \varnothing \vdash e_2 :^G T_2^G$ for some $\overline{CD^G} \vdash T_2^G \leq T_1^G$ and

- $e_2 \in$ Values, or
- $\exists\, e_3 \,.\, \overline{CD^G} \vdash e_2 \longrightarrow^G e_3$

This property states that, well-typed expressions of fully annotated programs neither produce a runtime error nor reach a stuck state (they are either evaluated to a value of the same type or another reduction could be applied).

Proof. If all the variables are annotated, then P^G is an FJ program, which is proved to be statically type safe in [25]. □

Property 2. *Progress of FC#^{G}.*

If $\vdash P^G = \overline{CD^G}\, e_1 :^G \diamond$ and $\overline{CD^G}, \varnothing \vdash e_1 :^G T_1^G$, then

- $e_1 \in$ Values, or
- $\overline{CD^G} \vdash e_1 \longrightarrow^G \mathit{error}$, or
- $\exists\, e_2 \,.\, \overline{CD^G} \vdash e_1 \longrightarrow^G e_2$

Proof. See Appendix D. □

The progress property indicates that a well-typed term is always evaluated to an object (value) or the runtime detects a type error (or another evaluation could be performed, for non-terminating programs). In other words, a well-typed term does not get stuck at some stage of its evaluation.

However, FC#$^{\mathrm{G}}$ does not provide preservation (or subject reduction), the second property required to make FC#$^{\mathrm{G}}$ type safe. The following type safety property is not fulfilled:

If $\vdash P^G = \overline{CD^G}\, e_1 :^G \diamond$ and $\overline{CD^G}, \varnothing \vdash e_1 :^G T_1^G$, then

- either $CD^G \vdash e_1 \longrightarrow^{G*} e_2$ and $\overline{CD^G}, \varnothing \vdash e_2 :^G T_2^G$ for some $\overline{CD^G} \vdash T_2^G \leq T_1^G$ and
  - ∘ $e_2 \in$ Values, or
  - ∘ $\exists\, e_3 \,.\, \overline{CD^G} \vdash e_2 \longrightarrow^G e_3$
- or $\overline{CD^G} \vdash e_1 \longrightarrow^G \mathit{error}$

Given the expression "`new C1().m().f`", if `m` returns `dynamic`, the type of the whole expression will also be `dynamic` (T-InvD$^{\mathrm{G}}$). If the body of `m` is "`new C2()`", the expression will be evaluated to "`new C2().f`". This new term may have a different type, or could even be ill-typed (if `C2` does not provide an `f` field) eventually producing an error at runtime. Thus, FC#$^{\mathrm{G}}$ is not type safe.

To make FC#$^{\mathrm{G}}$ type safe, the language semantics can be defined via translations, adding specific casts when `dynamic` expressions are used [2] or using reflection [18]. This makes the language type safe, but implies significant runtime performance penalties [10]. For this reason, we propose a program specialization mechanism that provides static type safety with better runtime performance.

$$\text{(S-UnionR}^{\text{S}}\text{)}\quad \frac{\exists\, i \in [1,n]\,.\,\overline{CD^S} \vdash C \leq C_i}{\overline{CD^S} \vdash C \leq C_1 \vee \ldots \vee C_n}$$

$$\text{(S-UnionL}^{\text{S}}\text{)}\quad \frac{\begin{array}{c}\overline{C_{sub}} = \{C_i \in C_1, \ldots, C_n\,.\,\overline{CD^S} \vdash C_i \leq C\} \qquad \#(\overline{C_{sub}}) \geq 1 \\ \forall\, C_{wrong} \in \{C_1, \ldots, C_n\} - \overline{C_{sub}}\,.\,warning(\text{“The type } C_{wrong} \text{ is not promotable to } C\text{”})\end{array}}{\overline{CD^S} \vdash C_1 \vee \ldots \vee C_n \leq C}$$

Figure 6: Additional subtyping rules for FC#$^{\text{S}}$.

3.2. FC#^{S}

We define FC#$^{\text{S}}$ as the output of the proposed specialization, receiving a FC#$^{\text{G}}$ program and translating it into FC#$^{\text{S}}$. The only syntactic difference with FC#$^{\text{G}}$ is related to types (the whole syntax can be consulted in Appendix E). In FC#$^{\text{S}}$, a type is defined as:

$$T^S \in \text{Type} \quad ::= \quad T^S \vee T^S \mid C$$

A type in FC#$^{\text{S}}$ can be a class or a union type containing more types [26], so `dynamic` is not a valid type in FC#$^{\text{S}}$. A union type $T_1 \vee T_2$ denotes the ordinary union of the set of values belonging to T_1 and the set of values belonging to T_2 [27], representing the least upper bound of T_1 and T_2 [28]. Union types hold the following properties:

$$T^S \vee T^S = T^S$$

$$T_1^S \vee T_2^S = T_2^S \vee T_1^S$$

$$T_1^S \vee (T_2^S \vee T_3^S) = (T_1^S \vee T_2^S) \vee T_3^S = T_1^S \vee T_2^S \vee T_3^S$$

3.2.1. FC#^{S} type system

Figure 6 shows the additional rules added to the existing subtyping relation. There is no $\lesssim$ relation in FC#$^{\text{S}}$, since `dynamic` is not a valid type. S-UnionR$^{\text{S}}$ denotes that, since a union type is the least upper bound of the types it collects, a class C is a subtype of any union type holding C.

The classical definition of union types states that a union type is a subtype of C when *all* the classes in the union type are subtypes of C [26]. S-UnionL$^{\text{S}}$ in Figure 6 shows a more lenient definition, since *at least* one ($\#(\overline{C_{sub}}) \geq 1$) class in the union type must be a subtype of C. A warning message is shown if there are one or more classes in the union type not fulfilling this predicate. Therefore, if warnings are considered as type errors, S-UnionL$^{\text{S}}$ will denote the classical definition: all the classes in the union type must be subtype of C (Lemma 3 in Appendix I).

Figure 7 shows the new typing rules added to FC#$^{\text{S}}$ in order to consider union types (the whole type system for FC#$^{\text{S}}$ can be consulted in Appendix F). The T-FieldU$^{\text{S}}$ rule types field access when the object is a union type ($C_1 \vee \ldots \vee C_n$). As discussed, (T-FieldU$^{\text{S}}$)

$$\frac{\begin{array}{c}\overline{CD^S}, \Gamma \vdash e :^G C_1 \vee \ldots \vee C_n \\ \overline{C_{withf}} = \{C_i \in C_1, \ldots, C_n \,.\, T_i^S f \in fields(\overline{CD^S}, C_i)\} \qquad n_f = \#(\overline{C_{withf}}) \geq 1 \\ \forall\, C_{wrong} \in \{C_1, \ldots, C_n\} - \overline{C_{withf}} \,.\, warning(\text{“Field } f \text{ not found in } C_{wrong}\text{”})\end{array}}{\overline{CD^S}, \Gamma \vdash e.f :^S T_1^S \vee \ldots \vee T_{n_f}^S}$$

(T-InvU$^{\text{S}}$)

$$\frac{\begin{array}{c}\overline{CD^S}, \Gamma \vdash e :^G C_1 \vee \ldots \vee C_n \qquad \overline{CD^S}, \Gamma \vdash \overline{e_{arg}} :^S \overline{T_{arg}^S} \\ \overline{C_{withm}} = \{C_i \in C_1, \ldots, C_n \,.\, type(method(\overline{CD^S}, C_i, m)) = \overline{T_{p_i}^S} \to T_{r_i}^S \text{ and } \overline{T_{arg}^S} \leq \overline{T_{p_i}^S}\} \\ n_m = \#(C_{withm}) \geq 1 \\ \forall\, C_{wrong} \in \{C_1, \ldots, C_n\} - \overline{C_{withm}} \,.\, warning(\text{“}C_{wrong} \text{ does not provide a suitable } m \text{ method”})\end{array}}{\overline{CD^S}, \Gamma \vdash e.m(\overline{e_{arg}}) :^S T_{r_1}^S \vee \ldots \vee T_{r_{n_m}}^S}$$

Figure 7: New typing rules for FC#$^{\text{S}}$.

at least one type in the union type must provide the f field. A warning message is shown for all the types in the union type not providing f. The type of the field access expression is inferred as a union type holding the types of the f fields ($T_1^S \vee \ldots \vee T_{n_f}^S$).

T-InvU$^{\text{S}}$ does the same for method invocation. When the implicit object in the invocation is a union type, at least one class in the union type must provide an appropriate m method ($\#(\overline{C_{withm}}) \geq 1$), where each argument is subtype of the corresponding parameter ($\overline{T_{arg}^S} \leq \overline{T_{p_i}^S}$). Warning messages are shown for those classes in the union type not fulfilling such condition. The type of the invocation is a union type with the return types of those appropriate m methods ($T_{r_1}^S \vee \ldots \vee T_{r_{n_m}}^S$).

As indicated in [29], this interpretation of union types gives FC#$^{\text{S}}$ the flavor of dynamic languages, allowing the utilization of flow-sensitive types [30]. In our motivating example, after specializing the program in Figure 1, T-InvU$^{\text{S}}$ will show a warning message in the third invocation to `DistanceToOrigin`, because `Triangle` does not provide `GetX` and `GetY` methods. For the fourth invocation, a compiler error is shown because no class in the union type provides the an appropriate `GetX` method.

3.2.2. FC#^{S} semantics

The semantics of FC#$^{\text{S}}$ is detailed in Appendix G. It is basically the same as FC#$^{\text{G}}$, but subtyping ($\leq$) is used instead of consistent subtyping ($\lesssim$). Additionally, we define the semantics with two reduction relations: $\longrightarrow^S$ represents evaluation without runtime type error checking, and $\longrightarrow^{SE}$ includes it. We distinguish these two relations to have the notion of potentially faster execution ($\longrightarrow^S$), where type errors do not need to be checked at runtime.

3.2.3. Properties of FC#^{S}

Property 3. *Progress of FC#^{S}.*

If $\vdash P^S = \overline{CD^S}\, e_1 :^S \diamond$ and $\overline{CD^S}, \varnothing \vdash e_1 :^S T_1^S$, then

– $e_1 \in$ Values, or

– $\overline{CD^S} \vdash e_1 \longrightarrow^{SE} error$, or

– $\exists\, e_2 \,.\, \overline{CD^S} \vdash e_1 \longrightarrow^{SE} e_2$

Proof. See Appendix H. □

As in FC#$^\mathrm{G}$, FC#$^\mathrm{S}$ is not type safe since the reduction derivations can change types of expressions at runtime (see a discussion about that in Lemma 8 in Appendix I). However, the progress property guarantees that a well-typed term can always be evaluated.

Property 4. *Static type safety of FC#^{S} when warnings are considered as errors.*

If $\vdash P^S = \overline{CD^S}\, e_1 :^S \diamond$ and $\overline{CD^S}, \varnothing \vdash e_1 :^S T_1^S$ without any warning, then

– either $e_1 \in$ Values,

– or $\exists\, e_2 \,.\, \overline{CD_G} \vdash e_1 \longrightarrow^S e_2$ and $\overline{CD_S}, \varnothing \vdash e_2 :^S T_2^S$ for some $\overline{CD_S} \vdash T_2^S \leq T_1^S$ without any warning

Proof. See Appendix I. □

With these two properties, we can see how FC#$^\mathrm{S}$ is statically type safe when warnings are considered as errors. If so, no runtime error is produced. Therefore, warnings tell the programmer which expressions in the source code could produce runtime errors. Moreover, FC#$^\mathrm{S}$ guarantees evaluation without getting stuck even with warnings (the progress property).

In the static type safety property (Property 4), we use the $\longrightarrow^S$ reduction instead of $\longrightarrow^{SE}$, meaning that many runtime checks are not required when a program has been compiled without warnings. This benefit, in addition to the optimized code generated (Section 5), provides better runtime performance of FC#$^\mathrm{S}$ programs.

4. Program specialization

As mentioned, the proposed program specializer takes a program written in FC#$^\mathrm{G}$ and generates a semantically equivalent one in FC#$^\mathrm{S}$. The process replaces `dynamic` types with type annotations, including union types (Figure 9 shows an example specialization of the code in Figure 1). The specialized program has the same semantics as the original one, it is statically type safe (when warnings are considered as errors), and performs fewer dynamic type checking operations to provide better runtime performance.

Our program specializer is a rule-based system, where rules have the following form: $P^G, \overline{CD^S_{in}}, \Gamma \vdash e \Rightarrow \langle e', \overline{CD^S_{out}} \rangle$. P^G is the FC#$^\mathrm{G}$ source program to be specialized, $\overline{CD^S_{in}}$ are the class definitions specialized so far, Γ is a typing context, e is an expression that is specialized to e', and $\overline{CD^S_{out}}$ are the class definitions specialized after specializing e.

The specialization process is the following one. First, P^G must be a well-typed FC#$^{\text{G}}$ program, i.e. $\vdash P^G = \overline{CD^G}\, e :^G \diamond$ and $\overline{CD^G}, \varnothing \vdash e :^G T^G$. Then, the input program P^G is then specialized to $P^S = \overline{CD^S}\, e'$, so $P^G, \varnothing, \varnothing \vdash e \Rightarrow \langle e', \overline{CD^S} \rangle$. The specialized program is semantically equivalent to the original one (Section 4.1), it can be checked if it is statically type safe (Section 3.2.3), and its execution performs fewer type checks at runtime (Section 3.2.3).

Figure 8 shows the specialization rules. For variables (SP-Var) and the `this` keyword (SP-This), the specializer maintains the original program with no changes in the specialized classes ($\overline{CD^S_{in}}$). For the field access expression (SP-Field), the object (e) is replaced with its specialization (e'), and the same f field is accessed in the specialized object. The specialized classes are those generated in the object specialization ($\overline{CD^S_{out}}$).

SP-InvC specializes a method invocation when the type of the specialized object is a class C_{obj} (not a union type). First, the implicit object e_{obj} is specialized to e'_{obj}. Second, the system specializes the arguments ($e_{arg_i} \Rightarrow \langle e'_{arg_i}, \overline{CD^S_i} \rangle$). In the specialization process, the output classes of one specialization are the input classes for the next specialization, and so forth. Therefore, the specialization of each argument considers the classes specialized in the specialization of both the implicit object and the previous arguments. The last step is the specialization of the m method, performed by the *specializem* function detailed in Appendix J. m is specialized to a new m_n method added to the specialized class of the implicit object (C_{obj}). That new m_n method replaces the `dynamic` parameters and return type of the original method m with the argument types inferred by the specializer (in FC#$^{\text{G}}$, `dynamic` is not a valid type). If the m method is overridden, *specializem* also specializes the m methods in the derived classes.

The specialized code in Figure 9 shows different examples of specializations performed by SP-InvC. The SP-InvC rule is applied in the first invocation to `DistanceToOrigin` in the `Main` method. For that particular invocation, a new version of the method is created (`DistanceToOrigin_1`), replacing the `dynamic` parameter type with the type of the argument (`Circumference`). The new `DistanceToOrigin_1` method is added to the specialized `Distance` class (Figure 9). The same specialization occurs for all the `GetX` and `GetY` methods in Figure 1 that return `dynamic`.

The specializer simulates the execution of the application, inferring type information of the evaluated expressions. Before specializing an invocation, the implicit object and the arguments are specialized (and hence their types are inferred). When a `dynamic` type is used in the input program (e.g., an argument or return value), its `dynamic` type annotation is replaced with the type inferred in the specialized code. The specialization of a method invocation requires the specialization of the method body (see *specializem* in Appendix J), so termination of the evaluation specialization process must be guaranteed (Property 5). If a method has already been specialized for some argument types, the *specializem* function returns the existing specialization (Appendix J).

(SP-Var)

$$\frac{}{P^G, \overline{CD^S_{in}}, \Gamma \vdash x \Rightarrow \langle x, \overline{CD^S_{in}} \rangle}$$

(SP-This)

$$\frac{}{P^G, \overline{CD^S_{in}}, \Gamma \vdash \texttt{this} \Rightarrow \langle \texttt{this}, \overline{CD^S_{in}} \rangle}$$

(SP-Field)

$$\frac{P^G, \overline{CD^S_{in}}, \Gamma \vdash e \Rightarrow \langle e', \overline{CD^S_{out}} \rangle}{P^G, \overline{CD^S_{in}}, \Gamma \vdash e.f \Rightarrow \langle e'.f, \overline{CD^S_{out}} \rangle}$$

(SP-InvC)

$$\frac{\begin{array}{c} P^G, \overline{CD^S_{in}}, \Gamma \vdash e_{obj} \Rightarrow \langle e'_{obj}, \overline{CD^S_0} \rangle \\ \overline{CD^S_0}, \Gamma \vdash e'_{obj} :^S C_{obj} \qquad n_{arg} = \#(\overline{e_{arg}}) \qquad \overline{P^G, \overline{CD^S_{i-1}}, \Gamma \vdash e_{arg_i} \Rightarrow \langle e'_{arg_i}, \overline{CD^S_i} \rangle}^{i \in [1, n_{arg}]} \\ \overline{\overline{CD^S_i}, \Gamma \vdash e'_{arg_i} :^S T^S_{arg_i}}^{i \in [1, n_{arg}]} \qquad \overline{CD^S_{out}}, n = \mathit{specializem}(P^G, \overline{CD^S_{n_{arg}}}, \Gamma, C_{obj}, m, \overline{T^S_{arg}}) \end{array}}{P^G, \overline{CD^S_{in}}, \Gamma \vdash e_{obj}.m(\overline{e_{arg}}) \Rightarrow \langle e'_{obj}.m\_n(\overline{e'_{arg}}), \overline{CD^S_{out}} \rangle}$$

(SP-InvU)

$$\frac{\begin{array}{c} P^G, \overline{CD^S_{in}}, \Gamma \vdash e_{obj} \Rightarrow \langle e'_{obj}, \overline{CD^S_0} \rangle \qquad \overline{CD^S_0}, \Gamma \vdash e'_{obj} :^S C_1 \vee \ldots \vee C_{n_u} \qquad n_{arg} = \#(\overline{e_{arg}}) \\ \overline{P^G, \overline{CD^S_{i-1}}, \Gamma \vdash e_{arg_i} \Rightarrow \langle e'_{arg_i}, \overline{CD^S_i} \rangle}^{i \in [1, n_{arg}]} \qquad \overline{\overline{CD^S_i}, \Gamma \vdash e'_{arg_i} :^S T^S_{arg_i}}^{i \in [1, n_{arg}]} \\ \overline{C_{n_u}} = \{C_1, \ldots, C_{n_u}\} \qquad n_{new} = \mathit{newmethod}(\overline{CD^S_{n_{arg}}}, \overline{C_{n_u}}, m) \qquad \overline{CD^{S\prime}_0} = \overline{CD^S_{n_{arg}}} \\ \overline{\overline{CD^{S\prime}_i} = \mathit{specializemn}(P^G, \overline{CD^{S\prime}_{i-1}}, \Gamma, C_{obj}, m, \overline{T^S_{arg}}, n_{new})}^{i \in [1, n_u]} \qquad \overline{CD^S_{out}} = \overline{CD^{S\prime}_{nu}} \end{array}}{P^G, \overline{CD^S_{in}}, \Gamma \vdash e_{obj}.m(\overline{e_{arg}}) \Rightarrow \langle e'_{obj}.m\_n_{new}(\overline{e'_{arg}}), \overline{CD^S_{out}} \rangle}$$

(SP-InvE)

$$\frac{\begin{array}{c} P^G, \overline{CD^S_{in}}, \Gamma \vdash e_{obj} \Rightarrow \langle e'_{obj}, \overline{CD^S_0} \rangle \\ n_{arg} = \#(\overline{e_{arg}}) \qquad \overline{P^G, \overline{CD^S_{i-1}}, \Gamma \vdash e_{arg_i} \Rightarrow \langle e'_{arg_i}, \overline{CD^S_i} \rangle}^{i \in [1, n_{arg}]} \\ \nexists T_{obj} \,.\, \overline{CD^S_0}, \Gamma \vdash e'_{obj} :^S T_{obj} \ \text{ or } \ \nexists T_{arg_i}{}^{i \in [1, n_{arg}]} \,.\, \overline{CD^S_i}, \Gamma \vdash e'_{arg_i} :^S T^S_{arg_i} \end{array}}{P^G, \overline{CD^S_{in}}, \Gamma \vdash e_{obj}.m(\overline{e_{arg}}) \Rightarrow \langle e'_{obj}.m(\overline{e_{arg}}), \overline{CD^S_{in}} \rangle}$$

(SP-New)

$$\frac{\begin{array}{c} \overline{CD^S_0} = \overline{CD^S_{in}} \qquad n_{arg} = \#(\overline{e^S_{arg}}) \qquad \overline{P^G, \overline{CD^S_{i-1}}, \Gamma \vdash e_{arg_i} \Rightarrow \langle e'_{arg_i}, \overline{CD^S_i} \rangle}^{i \in [1, n_{arg}]} \\ \overline{CD^S_{out}} = \mathit{specializec}(P^G, \overline{CD^S_{n_{arg}}}, \Gamma, C, \overline{e'_{arg}}) \end{array}}{P^G, \overline{CD^S_{in}}, \Gamma \vdash \texttt{new}\, C(\overline{e_{arg}}) \Rightarrow \langle \texttt{new}\, C(\overline{e'_{arg}}), \overline{CD^S_{out}} \rangle}$$

Figure 8: Program specialization rules.

SP-InvU specializes a method invocation when the type of the specialized implicit object (e'_{obj}) is a union type ($C_1 \vee \ldots \vee C_{n_u}$). We have an example in the `figure.GetX` invocation in `DistanceToOrigin_2` of Figure 9 (the type of `figure` is the union type `Circumference` $\vee$ `Rectangle`). In that case, we must specialize the `GetX` method in `Circumference` and `Rectangle`, but the new method version must have the same exact name (duck typing). Since `Circumference` already has a specialized version (`GetX_1`), the new method will be named `GetX_2` in both types. This is the main difference between SP-InvC and SP-InvU. The *newmethod* function in SP-InvU computes the number of a fresh m method for all the types in the union type $(\overline{C_{n_u}})$. Afterwards, *specializemn* specializes the m method in all the classes in the union type, receiving the new version number n_{new}.

Since FC#$^{\mathrm{S}}$ is type safe and FC#$^{\mathrm{G}}$ is not, specialized programs may be ill-typed. Therefore, SP-InvE rule specializes those method invocations that may have a type error in the generated program; i.e., the specialized implicit object or arguments are not well typed (the last premise in SP-InvE). In this way, FC#$^{\mathrm{G}}$ guarantees that any well-typed program can be specialized (Property 5). An example application of SP-InvE is the `DistanceToOrigin_4` method in Figure 9. Since the `Triangle figure` parameter has no `GetX` method, SP-InvE does not specialize the `figure.GetX()` expression (the m method is not replaced in SP-InvE). A compiler error will be produced, when the specialized FC#$^{\mathrm{S}}$ program is compiled.

SP-New specializes object construction. After specializing the arguments ($e_{arg_i} \Rightarrow \langle e'_{arg_i}, \overline{CD^S_i}\rangle$), the *specializec* function specializes the class, its constructor and its fields (Appendix J). As mentioned at the beginning of this section, the specialization process starts with an empty collection of output classes. Then, *specializec* adds a new version of the class when an object is created —thus, if a program never creates an instance of a class, that class will not be included in the specialized program. As with method invocation, if a constructor parameter is `dynamic`, *specializec* replaces `dynamic` with the inferred type of the argument (otherwise, the original type is kept). The same process is followed for fields. In the example code in Figure 9, the first line in `Main` creates a `Circumference`, passing three `int` arguments. SP-New specializes the `Circumference` class by declaring its constructor parameters and all its fields as `int`.

Constructors may be invoked at different times with different parameter types. One example is the `Node` class in Figure 9. Its constructor is invoked passing a `Circumference` parameter first, and then a `Rectangle`, both as first arguments. In that case, the specialization system assigns a union type (`Circumference` $\vee$ `Rectangle`) to the type of the first parameter and the corresponding `info` field, because it knows that it could hold either type at runtime (see the *specializekf* function in Appendix J).

4.1. Properties of the specializer

Property 5. *Well-typed FC#^{G} programs can always be specialized (and termination of the specialization is ensured).*

$\forall \vdash P^G = \overline{CD^G} e_1 :^G \diamond$ and $\overline{CD^G}, \varnothing \vdash e_1 :^G T_1^G \,.\, \exists\, e_2, \overline{CD^S}$ such that $P^G, \varnothing, \varnothing \vdash e_1 \Rightarrow \langle e_2, \overline{CD^S}\rangle$.

Proof. See Appendix L. □

Property 6. *Program specialization preservers semantics (i.e., the original program and its specialization are semantically equivalent).*

If $\vdash P^G = \overline{CD^G}\, e_1 :^G \diamond$ and $\overline{CD^G}, \varnothing \vdash e_1 :^G T_1^G$ and $P^G, \varnothing, \varnothing \vdash e_1 \Rightarrow \langle e_2, \overline{CD^S}\rangle$, then

```
public class Circumference {
  int x, y, radius;
  public int GetX_1() { return this.x; }
  public int GetY_1() { return this.y; }
  public int GetX_2() { return this.x; }
  public int GetY_2() { return this.y; }
  public Circumference(int x, int y, int radius) {
    this.x = x; this.y = y; this.radius = radius;
} }
public class Rectangle {
  double x, y, width, height;
  public double GetX_2() { return this.x; }
  public double GetY_2() { return this.y; }
  public double GetX_3() { return this.x; }
  public double GetY_3() { return this.y; }
  public Rectangle(double x, double y,
                 double width, double height) {
    this.x = x; this.y = y;
    this.width = width; this.height = height;
  }
}
class Triangle {
  int x1, y1, x2, y2, x3, y3;
  public Triangle(int x1, int y1, int x2, int y2,
                int x3, int y3) {
    this.x1 = x1; this.y1 = y1; …
} }
class Node {
  public Circumference∨Rectangle info;
  public Node next;
  public Node(Circumference∨Rectangle info,
            Node next) {
    this.info = info; this.next = next;
} }

class Distance {
  static double DistanceToOrigin_1(
                  Circumference figure) {
    return Math.Sqrt(Math.Pow(figure.GetX_1(), 2) +
                   Math.Pow(figure.GetY_1(), 2));
  }
```

```
static double DistanceToOrigin_2(
                  Circumference∨Rectangle figure) {
  return Math.Sqrt(Math.Pow(figure.GetX_2(), 2) +
                   Math.Pow(figure.GetY_2(), 2));
}
static double DistanceToOrigin_3(
                  Rectangle∨Triangle figure) {
  return Math.Sqrt(Math.Pow(figure.GetX_3(), 2) +
                   Math.Pow(figure.GetY_3(), 2));
}
static double DistanceToOrigin_4(Triangle figure) {
                            // compiler error
  return Math.Sqrt(Math.Pow(figure.GetX(), 2) +
                   Math.Pow(figure.GetY(), 2));
}
static Circumference∨Rectangle ClosestToOrigin_1(
                  Node list) {
  if (list == null) return null;
  Circumference∨Rectangle element = list.info;
  double min = DistanceToOrigin_2(element);
  while (list.next != null) {
    list = list.next;
    double distance = DistanceToOrigin_2(list.info);
    if (distance < min) {
      element = list.info;
      min = distance;
  } }
  return element;
}
static void Main() {
  Circumference cir = new Circumference(0,0,10);
  Rectangle rec = new Rectangle(1.2,3.7,1.0,2.0);
  Triangle tri = new Triangle(1, 2, -1, 3, 7, 2);
  DistanceToOrigin_1(cir);
  bool condition = new Random().NextDouble() >= 0.5;
  DistanceToOrigin_2(condition ? cir : rec);
  DistanceToOrigin_3(condition ? rec : tri);
  DistanceToOrigin_4(tri);
  Node list = new Node(rec, new Node(cir, null));
  Circumference∨Rectangle fig =
               ClosestToOrigin_1(list);
} }
```

Figure 9: The specialized program for the one in Figure 1.

– $\exists\, \sigma \,.\, \overline{CD^G} \vdash e_1 \longrightarrow^{G*} \sigma$ and $\overline{CD^S} \vdash e_2 \longrightarrow^{S*} \sigma$, or
– $\overline{CD^G} \vdash e_1 \longrightarrow^{G*} error$ and $\overline{CD^S} \vdash e_2 \longrightarrow^{SE*} error$, or
– $\exists\, e_3\, e_4 \,.\, \overline{CD^G} \vdash e_1 \longrightarrow^{G} e_3$ and $\overline{CD^S} \vdash e_2 \longrightarrow^{S} e_4$

Proof. See Appendix M. □

The first property ensures that the rule-based specialization system does not get stuck with well-typed FC#$^{\text{G}}$ programs. It also proves termination of the specialization process. Therefore, we demonstrate that any well-typed FC#$^{\text{G}}$ program can be specialized to another one written in FC#$^{\text{S}}$. Moreover, the second property shows that the result of evaluating a convergent FC#$^{\text{G}}$ program (i.e., a program that terminates) is the same as the evaluation of its specialization. Therefore, our program specialization system provides the following benefits:

1. The programmer is able to know whether a gradually typed program is statically type safe. This process is done in three steps. First, the FC#$^{\mathrm{G}}$ program is type-checked. Second, if the program is well-typed, it is then specialized. This second step is always possible, because Property 5 demonstrates that any well-typed FC#$^{\mathrm{G}}$ program can be specialized. The third step checks the static type safety of the specialized FC#$^{\mathrm{S}}$ code. Property 4 demonstrates that FC#$^{\mathrm{S}}$ is statically type safe when warnings are considered as errors. Therefore, a well-typed FC#$^{\mathrm{G}}$ program specialized into another one that compiles without warnings or errors is statically type safe.

2. If a FC#$^{\mathrm{G}}$ program is well-typed, the possible threats to type safety are shown as warnings, and they are caused by the use of `dynamic`. Since Property 1 proves that any well-typed program can be specialized, and Property 4 demonstrates static type safety when warnings are considered as errors, then warnings indicate potential threats to type safety. Property 1 states that fully annotated FC#$^{\mathrm{G}}$ programs are type safe, so the use of `dynamic` is the only threat to type safety.

3. If a well-typed FC#$^{\mathrm{G}}$ program is not statically type safe (warnings are shown), its execution does not get stuck. It can also be specialized and, if the specialized program is well-typed, it does not get stuck either. Property 2 demonstrates that any well-typed FC#$^{\mathrm{G}}$ program is an object or can be evaluated to another program or error (i.e., it does not get stuck). Property 5 proves that well-typed FC#$^{\mathrm{G}}$ programs can be specialized, and Property 3 that any well-typed FC#$^{\mathrm{S}}$ program does not get stuck.

4. Statically typed duck typing is provided. The use of `dynamic` allows flow-sensitive typing in FC#$^{\mathrm{G}}$, specializing `dynamic` to union types. Each dynamic type is specialized to another type. When an expression may have different types depending on the control flow, the specializer replaces `dynamic` with a union type collecting such different types. Property 6 demonstrates that the semantics of the original program (with `dynamic`) is the same as the specialized one (with union types). Therefore, FC#$^{\mathrm{G}}$ provides statically typed duck typing similar to languages with union types (e.g., Whiley or Pike), but without including that type constructor in the language (types are inferred and transparently added by the specializer).

5. When a program is compiled without warnings, many runtime type-checking operations are not required, providing better runtime performance. Property 4 demonstrates static type safety of FC#$^{\mathrm{G}}$ when warnings are considered as errors. Moreover, application execution does not need to perform many type-checking operations, already checked statically ($\longrightarrow^{SE}$ in Property 4). Section 6.2 shows an empirical assessment of the runtime performance benefits.

6. The type information added by the specialization process can be used to produce more efficient code. The specializer performs a kind of abstract interpretation to infer type information of dynamically typed code. The specializer uses union types

to represent all the different types a variable may hold. Section 5.2 shows how this information can be used to avoid the use of reflection, and hence generate more efficient code.

5. Language implementation

We have implemented the proposed system as part of a full-fledged programming language. We selected *StaDyn*, a gradually typed object-oriented language for the .NET Framework, created as an extension of C# [31]. It supports implicitly (and explicitly) typed references. Its syntax is the same as C#, but the *StaDyn* compiler gathers type information of `dynamic` references, improving compile-time type error detection and runtime performance [32]. Its source code is available for download at [33].

In the previous version of *StaDyn*, type inference was performed with a constraint-based type system [34]. This allowed the compiler to detect type errors over `dynamic` references, but reflection was used by the generated code. In this work, we have included in *StaDyn* the specialization mechanism proposed in this paper (Section 4). To compare both approaches, the following subsections describe the code generated for the old and new versions of the language. Although the compiler generates assembly code for the .NET platform, we show the generated code in C# for the sake of legibility.

5.1. Code generation for FC#^{G}

The *StaDyn* compiler generates code following the typical implementation of object-oriented gradually typed languages [18] for the .NET platform such as Visual Basic, C#, Boo and Cobra (see Section 6.1.1). We describe this approach by specifying it as a translation of FC#$^{\mathrm{G}}$ into C#. We focus on the most representative code generation templates (the rest of templates are detailed in Appendix N).

Given an e expression, $[\![e]\!](\overline{CD^G},\Gamma)$ represents the .NET code generated for that expression in a program with $\overline{CD^G}$ classes and the Γ type environment. For field access expressions, when the type of the object is statically known, the C# code is similar to FC#$^{\mathrm{G}}$ (Appendix N). However, when the object is `dynamic`, the compiler has no information about its type. In this case, reflection is used to get the type of the object and access its f field; if f is not provided, a runtime error is produced. That runtime behavior is provided by `Reflection.GetField` (its implementation is detailed in Appendix N).

$$\frac{\overline{CD^G},\Gamma \vdash e :^G dynamic}{[\![e.f]\!](\overline{CD^G},\Gamma) = \texttt{Reflection.GetField(}[\![e]\!](\overline{CD^G},\Gamma)\texttt{, "f")}}$$

Since the .NET assembly code does not provide a `dynamic` type, we translate `dynamic` references into `object` (as the C# compiler). To this aim, we define $|T|$ as the erasure of the type T [19]. In the generated code, we will use type erasures, since the original types (`dynamic` in FC#$^{\mathrm{G}}$ and union types in FC#$^{\mathrm{S}}$) are not supported.

$|\texttt{dynamic}| = \texttt{object}$
$|C| = C$

Gradually typed languages provide the implicit conversion of `dynamic` into any other type (the subtyping consistency relation defined in Section 3.1.2). Therefore, the generated code must add an explicit cast where this conversion takes place. For this purpose, we define the following *cast* function:

$$cast(T_1^G, T_2^G) = \begin{cases} \texttt{(}C\texttt{)} & \text{if } |T_1^G| = \texttt{object} \text{ and } |T_2^G| = C \\ \texttt{/* nothing */} & \text{otherwise} \end{cases}$$

With this function, we can generate code for method invocations, when the type of the implicit object is statically known. We simply add to each argument a cast to the parameter type, when the argument is `dynamic` and the parameter is not:

$$\frac{\overline{CD^G}, \Gamma \vdash e :^G C \qquad type(method(\overline{CD^G}, C, m)) = \overline{T_p^G} \rightarrow T_r^G \qquad \overline{CD^G}, \Gamma \vdash \overline{e_{arg}} :^G \overline{T_{arg}^G}}{[\![e.m(\overline{e_{arg}})]\!](\overline{CD^G}, \Gamma) = [\![e]\!](\overline{CD^G}, \Gamma)\texttt{.m(}\overline{cast(T_{arg}^G, T_p^G)[\![e_{arg}]\!](\overline{CD^G}, \Gamma)}\texttt{)}}$$

Similar to field access, when the implicit object in a method invocation is `dynamic`, reflection is used to invoke a method:

$$\frac{\overline{CD^G}, \Gamma \vdash e :^G dynamic \qquad \overline{CD^G}, \Gamma \vdash \overline{e_{arg}} :^G \overline{T_{arg}^G}}{\begin{array}{l}[\![e.m(\overline{e_{arg}})]\!](\overline{CD^G}, \Gamma) = \\ \quad \texttt{Reflection.Invoke(}[\![e]\!](\overline{CD^G}, \Gamma)\texttt{, "m", new object[]\{}\overline{[\![e_{arg}]\!](\overline{CD^G}, \Gamma)}\texttt{\})}\end{array}}$$

5.2. Code generation for FC#^{S}

The previous code generation scheme is followed by most gradually typed languages. In this subsection, we describe how the type information gathered by the specializer can be used to generate more efficient code. In FC#^{S}, no `dynamic` reference exists, but we have union types. Therefore, we define type erasure of union types as $|C_1 \vee \ldots \vee C_n| = \texttt{object}$.

We only describe the code generation rules for field access and method invocation, when the implicit object is a union type (the rest of rules are detailed in Appendix O). This is the rule for field access:

$$\frac{\begin{array}{cc}\overline{CD^S}, \Gamma \vdash e :^S T_e^S & T_e^S = C_1 \vee \ldots \vee C_n \\ \overline{C_{union}} = \{C_i \in C_1, \ldots, C_n \,.\, f \in fields(\overline{CD^S}, C_i)\} & n_u = \#(\overline{C_{union}})\end{array}}{[\![e.f]\!](\overline{CD^S}, \Gamma) =}$$

```
    if $n_u == 1$ then
        ($cast(T_e^S, C_{union_1})$[[e]]($\overline{CD^S}$, $\Gamma$)).f
    else
        for $i = 1$ to $n_u - 1$ do
            if $i == 1$ then _temp=[[e]]($\overline{CD^S}$, $\Gamma$)
            else _temp
            is $C_{union_i}$ ? ($cast(T_e^S, C_{union_i})$_temp).f :
        ($cast(T_e^S, C_{union_{n_u}})$_temp).f
```

We first check how many types in the union type provide the f field. There must be at least one type, since we only generate code for well-typed programs. If only one type provides the f field, a simple cast is done. Otherwise, a sequence of nested type inspections are produced using the `?:` ternary operator. We evaluate whether the dynamic type of the expression is one type in the union type, using the `is` operator. If the expression has the expected type, the field is accessed after casting the object; otherwise, we return the value of another nested ternary expression. Notice that we use a temporary `_temp` variable (Appendix O) to evaluate the implicit object only once.

As mentioned, when the implicit object is a union type, there may be types in the union type that do not provide the f field. In that case, the generated code only checks the types that provide the f field, ignoring the remaining types in the union type. This optimization provides a runtime performance benefit (in addition to avoid the use of reflection).

Method invocation is very similar to field access. The only additional consideration is that union-typed arguments require a cast to non-union-type parameters because the type erasure for union types is `object`:

$$\frac{\begin{array}{ll} \overline{CD^S}, \Gamma \vdash e :^S T^S_e & T^S_e = C_1 \vee \ldots \vee C_n \\ \multicolumn{2}{l}{\overline{C_{union}} = \{C_i \in C_1, \ldots, C_n \,.\, type(method(\overline{CD^S}, C_i, m)) = \overline{T^S_{p_i}} \to T^S_{r_i} \text{ and } \overline{T^S_{arg}} \leq \overline{T^S_{p_i}}\}} \\ n_u = \#(\overline{C_{union}}) & \overline{CD^S}, \Gamma \vdash \overline{e_{arg}} :^S \overline{T^S_{arg}} \end{array}}{}$$

```
$[\![e.m(\overline{e_{arg}})]\!](\overline{CD^S},\Gamma) =$
    if $n_u == 1$ then
        ($cast(T^S_e, C_{union_1})[\![e]\!](\overline{CD^S},\Gamma)$).m($\overline{cast(T^S_{arg}, T^S_{p_1})[\![e_{arg}]\!](\overline{CD^S},\Gamma)}$)
    else
        for $i = 1$ to $n_u - 1$ do
            if $i == 1$ then _temp=$[\![e]\!](\overline{CD^S},\Gamma)$
            else _temp
            is $C_{union_i}$ ? ($cast(T^S_e, C_{union_i})$_temp).m($\overline{cast(T^S_{arg}, T^S_{p_i})[\![e_{arg}]\!](\overline{CD^S},\Gamma)}$) :
        ($cast(T^S_e, C_{union_n})$_temp).m($\overline{cast(T^S_{arg}, T^S_{p_{nu}})[\![e_{arg}]\!](\overline{CD^S},\Gamma)}$)
```

6. Evaluation

In this section, we measure runtime performance, memory consumption and compilation time of the proposed specialization method added to the *StaDyn* language [31]. We first describe the evaluation methodology. Then, we present the results and discussions.

6.1. Methodology

The methodology comprises a description of the selected languages to be compared with *StaDyn*, the benchmark suites used in the evaluation, and a description of how data are measured and analyzed [32].

6.1.1. Programming languages

We compare *StaDyn* with the most widely used gradually typed languages for the .NET Framework, all of them compiled with their maximum optimization options. We use the same target platform, since our objective is to measure the influence of program specialization on the efficiency of the generated code. Afterwards, we compare our language with the state-of-the-art techniques for optimizing dynamically typed code.

These are the .NET gradual typing languages evaluated:

– C# 7.2. This version of C# combines static and dynamic typing. Its back-end is the DLR (Dynamic Language Runtime), released as part of the .NET Framework 4+. The DLR is a new layer over the CLR (Common Language Runtime) that provides a set of services to facilitate the implementation of dynamic languages [35].

– Visual Basic (VB) 2017. The VB programming language also supports gradual typing [36]. A dynamic reference is declared with the `Dim` reserved word, without setting a type. With this syntax, the compiler does not gather any type information statically, and type checking is performed at runtime.

– Boo 0.9.7. An object-oriented programming language for the CLI (Common Language Infrastructure) with Python inspired syntax. It is statically typed, but it also provides dynamic typing by using its special `duck` type [37].

– Fantom 1.0.70. Fantom is an object-oriented programming language that generates code for the Java VM, the .NET platform, and JavaScript. It is statically typed, but it provides the dynamic invocation of methods with the specific `->` message-passing operator [38].

– Cobra 0.9.6. Cobra is an object-oriented gradually typed programming language that provides compile-time type inference [39]. As C#, dynamic typing is provided with a distinctive `dynamic` type.

– IronPython 2.7.7. An open-source implementation of the Python programming language which is tightly integrated with the .NET Framework, targeting the DLR. It compiles Python programs into IL (Intermediate Language) bytecodes [40]. IronPython is a fully dynamically typed language. We have included it in the evaluation of code with no type annotations (Section 6.2.1) because it is an efficient implementation of a dynamic language for the .NET framework [41].

– *StaDyn* 2.1 and 2.0. [42]. Version 2.1 of *StaDyn* incudes the program specialization mechanism described in this article. We compare it with the previous version, 2.0, which does not perform program specialization. However, *StaDyn* 2.0 includes other important optimizations such as SSA transformations [43], type inference of dynamically typed code [34], and the utilization of the DLR runtime cache for long-running applications [16].

Some existing dynamic languages implement modern optimization techniques to significantly improve the runtime performance of dynamically typed code. Although these languages are not compiled to the .NET platform, we compare their state-of-the art optimizations with the program specialization approach proposed in this article:

- PyPy 2.7. PyPy is an alternative implementation of Python that provides JIT compilation, memory usage optimizations, and full compatibility with CPython [44]. PyPy implements a tracing JIT compiler to optimize program execution at runtime, generating dynamically optimized machine code for the hot code paths of commonly executed loops [44]. PyPy has been evaluated as the fastest Python 2 implementation [45]. To compare the benefits of PyPy with its reference implementation, we also measure CPython 2.7.14 in some tests.

- V8 6.5.73, the Google's open source JavaScript engine used in Chrome and Node.js [46]. V8 implements a runtime adaptive JIT compiler that dynamically optimizes the generated code based on heuristics of the code execution profile [46].

- SpiderMonkey 24.4. SpiderMonkey is the JavaScript engine of Mozilla, currently added to the Firefox Web browser and the GNOME 3 desktop [47]. IonMonkey is the name of the JIT compiler included in SpiderMonkey. IonMonkey implements many optimizations such as function inlining, linear-scan register allocation, type specialization, and loop-invariant code motion [47]. We measure the standalone distribution of SpiderMonkey 24 with and without IonMonkey in order to compare the impact of IonMonkey optimizations in the base JavaScript engine.

6.1.2. Benchmarks

We selected a collection of different applications to measure execution and compilation time, and memory consumption for each language implementation:

- Two well-known dynamically typed benchmarks:
  - Pybench. A Python benchmark designed to measure the performance of standard Python implementations [48]. Pybench is composed of a collection of tests measuring different aspects of the Python dynamic language.
  - Pystone, the Python version of the Dhrystone benchmark [49]. Pystone is commonly used to compare different implementations of the Python programming language.

- A subset of the statically typed Java Grande benchmark implemented in C# [50], including kernel programs and large scale applications:
  - Section 2 (Kernels). *FFT*, one-dimensional forward transformation of n complex numbers; *Heapsort*, the heap sort algorithm over arrays of integers; and *Sparse*, management of an unstructured sparse matrix stored in compressed-row format with a prescribed sparsity structure.
  - Section 3 (Large Scale Applications). *RayTracer*, a 3D ray tracer of scenes that contain 64 spheres, and are rendered at a resolution of 25x25 pixels.

- *Points*, a C# hybrid static and dynamic typing program designed to measure the performance of hybrid typing languages [34]. It computes different properties of two and three dimensional points.

We took Python (Pybench and Pystone) and C# (Java Grande and Points) programs, and manually translated them into the rest of languages. Although this translation might introduce a bias on the runtime performance of the translated programs, we have thoroughly checked that the same operations were executed in all the implementations. We have also verified that the benchmarks compute the same results in all the programs.

6.1.3. Data analysis

We have followed the methodology proposed in [51] to measure the runtime performance of programming languages. A two-step methodology is followed:

1. We measure the elapsed execution time of running the same program multiple times. This results in p (we have taken $p = 30$) measurements x_i with $1 \leq i \leq p$.
2. The confidence interval for a given confidence level (95%) is computed to eliminate measurement errors that may introduce a bias in the evaluation. The confidence interval is calculated using the *Student's t*-distribution because we took $p = 30$ [52].

In the subsequent figures, we show the mean of the confidence interval and indicate with bar whiskers the width of the confidence interval relative to the mean. If two confidence intervals do not overlap, we can conclude that there is a statistically significant difference with a 95% probability [51].

All the tests were carried out on a 2.13 GHz Intel Core 2 Duo P7450 system with 4 GB of RAM, running an updated 64-bit version of Windows 10 and the .Net Framework 4.7.1. The benchmarks were executed after system reboot, removing the extraneous load, and waiting for the operating system to be loaded. To compute average percentages, factors and orders of magnitude, we use the geometric mean.

6.2. Execution time

We first measure execution times of programs with no type annotations to see how the different languages optimize dynamically typed code. Then, we compare execution times when all the variables are explicitly typed (annotated), measuring runtime performance of statically typed code.

6.2.1. Code with no type annotation

All the programs measured in this subsection have no type annotation, declaring all the variables, fields, method parameters and return values as `dynamic`. We first compare execution time of those languages compiled for the .Net platform. Figure 10 presents the average execution time relative to *StaDyn* without program specialization (*StaDyn*-SP refers to *StaDyn* 2.1, which includes the specializer). Figure 10 shows how program specialization achieves an average runtime performance gain of 419% compared to the previous version of *StaDyn*. For Pybench and the two sections of Java Grande, the specialized code requires 33%, 5% and 2% the execution time used by *StaDyn*, respectively. The specializer obtains lower benefits for the two remaining benchmarks. Pystone performs most of its computation using local variables instead of method parameters. Since the previous version of *StaDyn* manages to infer the types of local variables [43], program specialization has a lower impact on the performance gain (35.14%). The Points program uses flow sensitive types, so the specialized program utilizes several union types (Section 3.2), requiring runtime type checking as described in Section 5.2.

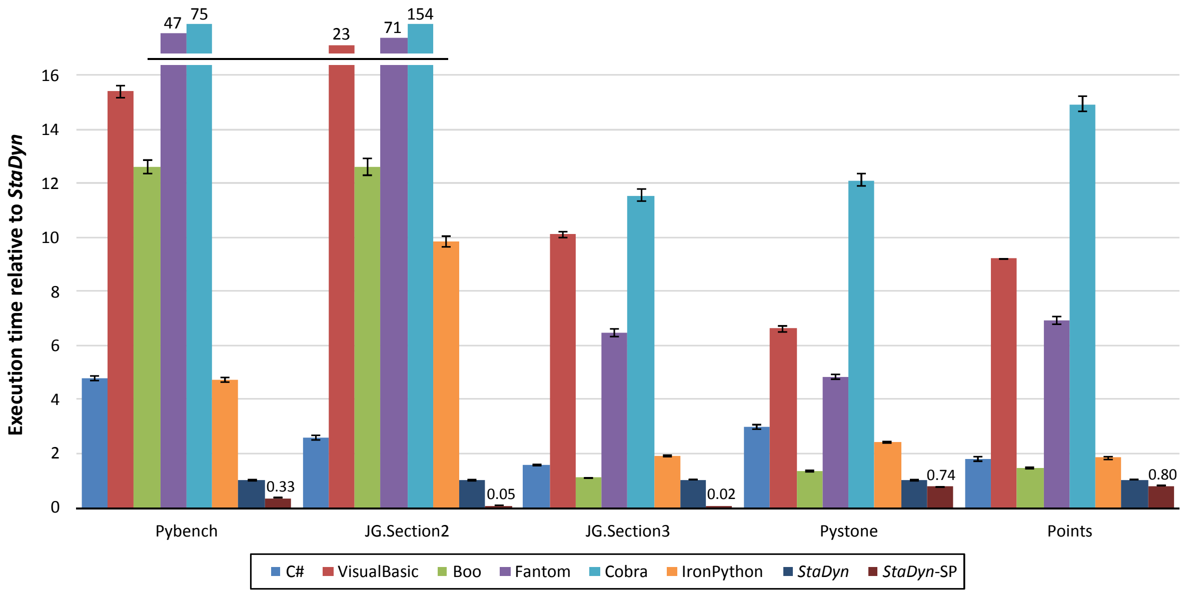

Figure 10: Execution times of .NET languages, relative to *StaDyn*.

In Figure 10, we can see that the specialized programs outperform all the gradually typed languages compiled for the .NET platform. On average, *StaDyn*-SP is 13.1, 16.6, 17.1, 76.8 and 155.3 factors faster than C#, Boo, IronPython, Fantom and Cobra, respectively. Our static program specializer provides important performance benefits compared to the runtime cache optimizations supported by the DLR (used by C# and IronPython) and the type inference techniques implemented by *StaDyn*, Boo and Cobra.

After comparing the runtime performance of gradually typed languages for the .NET platform, we now measure highly optimized implementations of dynamically typed languages to compare the existing state-of-the-art optimizations with our approach. Figure 11 shows how the specialized *StaDyn* programs perform faster than the rest of approaches. Our system provides 273%, 141% and 93% higher runtime performance than PyPy, SpiderMonkey with the IonMonkey optimizations and V8, respectively.

The proposed program specialization also achieves the highest performance gain relative to its base implementation (4.19 factors relative to *StaDyn* and 13.12 factors relative to C#). The tracing JIT compiler implemented by PyPy provides a 116% performance gain relative to CPython. For JavaScript, IonMonkey and V8 showed 57.7% and 96.6% performance improvement compared to SpiderMonkey without the IonMonkey optimizations.

6.2.2. Fully type-annotated code

We also evaluate source code that uses no dynamically typed reference, that is, programs with all the type annotations. In this case, no program specialization takes place, and the languages perform no optimizations of dynamically typed code. The Points application has not been evaluated, because it requires dynamic typing [34]. Python (IronPython, CPython and PyPy) and JavaScript (SpiderMonkey, IonMonkey and V8) implementations are not included because they do not provide a standard type-annotation mechanism.

Figure 12 shows the average execution time for this kind of code. As expected, both versions of *StaDyn* showed the same runtime performance, because no code is specialized. C# provides

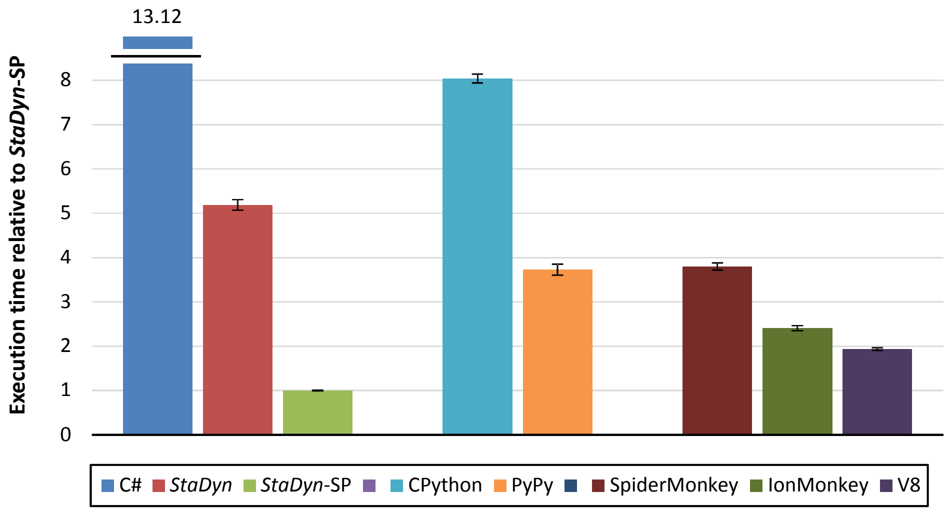

Figure 11: Execution times of highly optimized dynamic languages relative to *StaDyn*-SP.

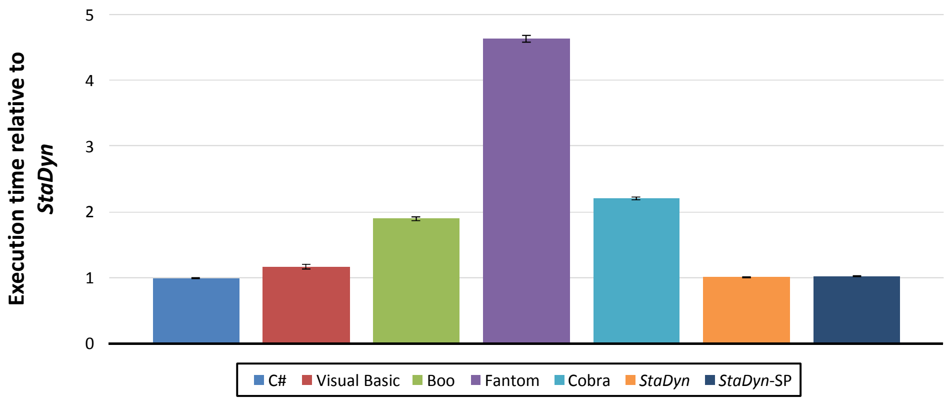

Figure 12: Execution times of fully type-annotated programs, relative to *StaDyn*.

the best runtime performance, only 2.5% higher than *StaDyn*. The C# commercial compiler provides more optimizations for statically typed code, causing slightly lower execution times. *StaDyn* provides better runtime performance than the other languages for fully type-annotated code: it is 16%, 89%, 120% and 363% faster than Visual Basic, Boo, Cobra and Fantom. These results show that, besides the important optimizations for dynamically typed code, *StaDyn* also generates efficient code for statically typed programs.

6.3. Memory consumption

We have measured the memory consumed at runtime by the generated code. Figure 13 shows the average memory consumptions for all the languages measured (Section 6.1.1), when all the programs have no type annotation (fully dynamically typed code). An important observation is that the *StaDyn* specializer adds no additional memory consumption to *StaDyn*: the average difference (0.82%) is lower than the confidence interval (1.17%), so this difference is not statistically significant [51]. Therefore, the supplementary methods generated by the specializer consume negligible additional memory at runtime.

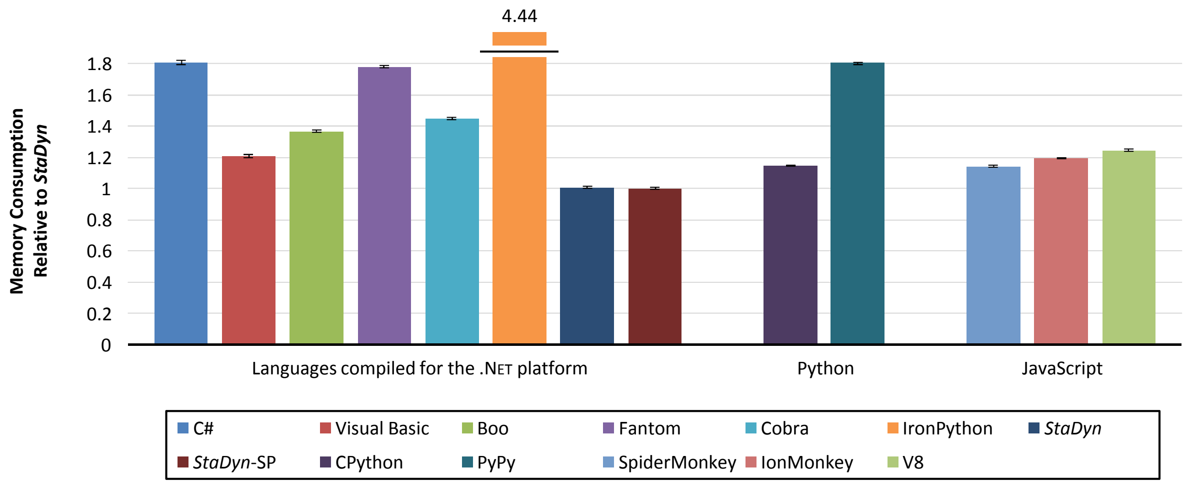

Figure 13: Memory consumption of the generated code, relative to *StaDyn* (no type annotation).

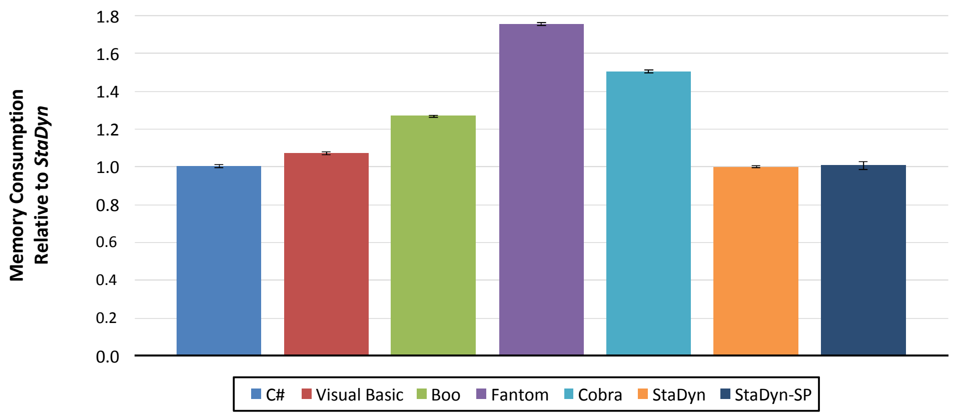

Figure 14: Memory consumption of the generated code, relative to *StaDyn* (fully type-annotated).

Figure 13 also shows that *StaDyn* is the programming language that requires fewer memory resources at runtime, because program specialization is performed statically. On the contrary, highly optimized implementations, such as PyPy, IonMonkey and V8, perform all the optimizations dynamically, consuming additional memory resources. C# and IronPython use the runtime cache of the DLR, and Fantom implements its own cache, causing significantly higher memory consumption.

Figure 14 shows the same comparison as Figure 13, when all the programs are fully type-annotated. C#, *StaDyn* and *StaDyn*-SP consume the same memory (no statistically significant difference) because they generate almost the same code when all the variables are explicitly typed. Visual Basic, Boo, Cobra and Fantom consume 7.4%, 27%, 51% and 75% more memory resources because of their runtime, not present in C# and *StaDyn*.

6.4. Compilation time

We have seen how program specialization provides significant runtime performance benefits (Section 6.2), and consumes no additional memory resources (Section 6.3). These benefits are provided because, unlike the other existing approaches, specialization is performed at compile

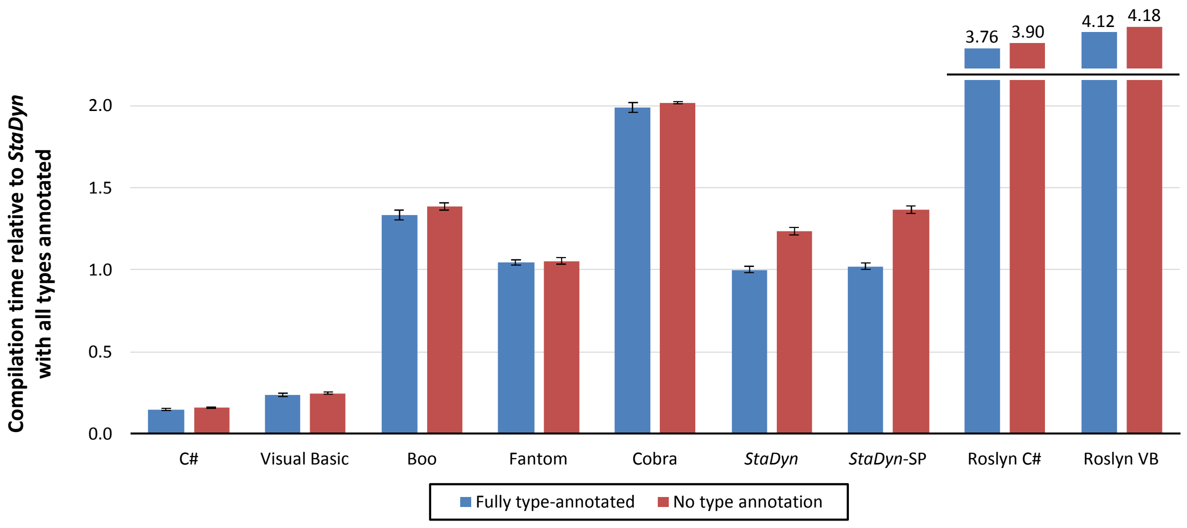

Figure 15: Compilation time relative to *StaDyn* fully typed-annotated.

time. Therefore, static program specialization implies a cost in compilation time. In order to evaluate that cost, we measure compilation time of the compiled languages identified in Section 6.1.1.

The *StaDyn*, Boo, Cobra and Fantom compilers are implemented in .NET (*StaDyn* and Boo are coded in C#, whereas Cobra and Fantom are implemented in their own languages). On the other hand, the C# and VB compiler implementations are native. Although these two production compilers have been included in our evaluation, we have also measured the two .NET versions released by Microsoft, called Roslyn [53]. The Roslyn compilers implement the C# and VB languages on .NET, exposing the compilers as services.

Figure 15 shows average compilation times for each language and compiler implementation. When programs are fully type-annotated, the specializer only requires 2% more average compilation time. This value is increased to 11.75% when programs have no type annotation, since many methods and fields are specialized at compile time. As shown in Table 1, the compiler generates 81 additional methods (out of 97) specialized by the compiler. The 11.75% extra compilation time is used to generate these 81 methods, producing an average performance optimization of 149% (Section 6.2.1).

If we compare compilation time of all the languages, the native compiler implementations of C# and VB are the fastest ones (Figure 15), requiring 15.4% and 24% the compilation time employed by *StaDyn*. However, if we compare the compilers implemented on the same platform as *StaDyn* (.NET), the *StaDyn* specializing compiler is 130% and 140% faster than Roslyn C# and Visual Basic, respectively. In fact, Fantom is the only .NET compiler that is faster than *StaDyn*-SP, just for fully dynamic programs (29.71%). The rest of .NET compilers (Boo, Cobra, Roslyn C# and Roslyn Visual Basic) require more compilation time.

Table 1 summarizes the number of specialized methods per program. The methods with no parameters are not specialized, and those with one parameter or more are specialized at least once. If a method is specialized, the original version is maintained in case it is invoked from another program. Table 1 shows how the generated programs have at least 41% of their methods specialized and produce significant performance benefits, which are always greater than the compilation time increase.

Benchmark	Comp. time increase	Performance gain	Original Methods No params	Original Methods Params (≥ 1)	Specialized methods	Total methods generated
Pybench	3.2%	201%	2	11	11	24
HeapSort	15.9%	4,710%	0	3	6	9
FFT	60.7%	558%	0	9	10	19
SparseMatmult	4.4%	2,079%	0	3	3	6
RayTracer	10.6%	3,790%	17	27	31	75
Pystone	31.0%	35%	5	12	12	29
Points	7.0%	25%	0	8	8	16

Table 1: Summary of specialized methods per benchmark, and the corresponding compilation time and runtime performance increases.

6.5. Discussion

Our evaluation shows how the proposed program specialization provides a significant runtime performance optimization with no additional memory consumption. The specializer simulates the execution of the application as a kind of abstract interpretation, inferring type information of dynamically typed code. The inferred types of the implicit object and arguments are used to specialize method invocations, field accesses and class definitions. However, if `dynamic` is used to interoperate with dynamically typed languages, the proposed program specialization would not be able to optimize the code (i.e., no abstract interpretation is performed).

The actual use of dynamic types in gradually typed languages depends on the programmer, the language, and the kind of application to be developed. Thus, we have studied the use of `dynamic` in C# by analyzing the existing C# projects in GitHub, using Google's BigQuery [54]. Out of 17 million C# source files, the `dynamic` type was used in 149.124 archives. To know how `dynamic` is used by C# programmers, we analyzed a sample of 200 random files. Each file in the sample belongs to a different programmer and project in order to avoid dependency on specific programming styles.

In 79.5% of the cases, `dynamic` was used to obtain duck typing; that is, to postpone compile-time type-checking until runtime. This kind of code could have been written with compile-time type annotations, but the programmer decided to disable static type-checking. In 18% of the files, `dynamic` was used to build `ExpandoObjects`, which are objects whose members can be dynamically added and removed at runtime. Although FC#^{G} does not provide such feature, our specialization technique could be applied to optimize `ExpandoObjects`. The 2.5% remaining files used `dynamic` to interact with other dynamically typed languages (IronPython and IronRuby).

Therefore, for the particular case scenario of C#, the use of `dynamic` only represents 0.88% (149.124 files out of 17 million) of the code written in that language. However, for those programs using `dynamic`, our proposed system could optimize 97.5% of such programs.

7. Related work

7.1. Gradual typing

Since both dynamic and static typing offer important benefits, there have been many works aimed at obtaining the advantages of both approaches in the very same programming language. One of the first works was soft typing [55], which applied static typing to a dynamically typed language such as Scheme. Soft typing does not control which parts in a program are statically checked, and static type information is not used to optimize the generated code. The approach proposed by Abadi *et al.* [56] adds a `Dynamic` type to lambda calculus, including two conversion operations (`dynamic` and `typecase`), generating a verbose code deeply dependent on its dynamism.

The works of quasi-static typing [57], hybrid typing [58] and gradual typing [7] perform implicit conversions between dynamic and static code, employing subtyping relations in the case of quasi-static and hybrid typing, and the consistency relation in the case of gradual typing. Gradual typing was first defined with the $\lambda^{?}_{\rightarrow}$ functional calculus [7] but it has also been defined for object-based languages, showing that gradual typing and subtyping are orthogonal and can be combined [2]. Gradual typing has also been integrated with ownership types [59], refinement types [60], session types [61], type inference [62], and union and intersection types [63].

Sound gradually typed languages insert runtime checks to achieve type soundness for the overall program. Therefore, the programmer can rely on the language implementation to provide meaningful error messages at runtime. However, these runtime type checks imply a significant performance overhead [10]. Rastogi *et al.* evaluated the runtime performance of sound gradual typing using Safe TypeScript [64]. The average performance cost for dynamic code with no type annotations was 22 factors [64]. For this reason, there have recently been works focused on optimizing the implementation of sound gradually typed languages.

Siek *et al.* propose monotonic references to avoid the runtime overhead of dynamic typing in statically typed regions [65]. Herman *et al.* propose the reduction of type checking operations by combining adjacent type coercions, providing potential optimizations in space and time [66]. However, these works are theoretical and no implementation and performance evaluation exists yet.

Pycket implements a tracing JIT compiler for the gradually typed Racket language, which supports features such as contracts, continuations classes, structures and dynamic binding [13]. Pycket is implemented in the RPython meta-tracing framework, originally created for PyPy. RPython automatically generates tracing JIT compilers from interpreters [67]. Among other optimizations, Pycket performs runtime type specialization of different data structures. The tracing JIT compiler provided by RPython has allowed Pycket to eliminate more than 90% of the gradual typing overhead introduced by Typed Racket [68]. However, it seems that there is still room for further optimization [69, 68].

7.2. Automatic type annotation

There exist some works focused on the automatic annotation of types in gradually typed programs. Migrational typing automatically migrates a program to be as static as possible, introducing the least number of dynamic types necessary to remove a type error [70]. Migrational typing conceptually types the whole migration space, marking where type errors occur so that it can safely present the possible migrations for the program. To know the possible types a variable may have, migrational typing uses constraint generation and unification algorithms for a functional language [70]. The algorithm scales linearly with program size.

Pytype is a Google's open source project aimed at providing static type checking of Python programs [71]. Pytype is capable of analyzing existing Python code with no type annotations, and detects some type errors at compile time. It also supports type annotations following the PEP 484 specification included in Python 3.5+ [72], and provides a mechanism to specify types of existing modules in `.pyi` files [73]. All the types of the standard Python library are included in a repository called typeshed. Additionally, Pytype is able to analyze Python programs without types, generate `.pyi` files with type specifications, and annotate the source code with the inferred types [74].

stypy is another tool to perform static type checking of Python programs with no type annotation [75]. *stypy* translates one program into Python code that type-checks the original code. The generated type checker detects type errors in different tricky Python idioms, and ensures termination [76]. Types are inferred with a kind of abstract execution, where *stypy* evaluates expression types instead of their values. Although the inferred type information is only used to detect type errors before program execution, it is also planned to use it for automatic type annotation [76].

Another different approach is the utilization of machine learning to annotate types in gradually typed code, learning type annotations from large sets of programs [77]. JSNice is a tool that uses Condition Random Fields (CRF) to annotate types in JavaScript code and deobfuscate variable names. JSNice uses the type annotations included in different versions of the language to train the CRF model. Experimentally, JSNice is able to predict type annotations correctly in 81% of the cases [77]. They only predict built-in types, not considering user-defined ones.

7.3. Program specialization

Partial evaluation, also called program specialization, is a program optimization technique that produces new programs that run faster than the original ones, guaranteeing to behave in the same way [78]. The optimization is based on precomputing all the static input at compile time, and generating another program where that precomputation is not performed at runtime, causing a runtime performance gain. Yoshihiko Futamura identified a mechanism, known as the Futamura projections, to obtain compilers from interpreters by applying partial evaluation [79]. Partial evaluation has been used to optimize different applications such as ray tracers, pattern recognizers, training of neural networks and scientific computation [78].

Ulrik Schultz presents a formal description of partial evaluation for an extension of Featherweight Java with integer and Boolean expressions and type casts [80]. Programs are specialized to other ones, providing better runtime performance. The code is specialized by propagating the static values known at compile time, and using them to reduce field lookups, method invocations, and non-object computations. The author does not prove the soundness of the proposed system, and termination is not ensured [80]. The formal description was used to implement JSpec, a complete partial evaluator for Java programs [81].

Runtime program specialization is currently used to optimize dynamic languages. Psyco is a just-in-time Python specializer that uses actual runtime values and types to improve runtime performance [82]. The key idea of Psyco is that program specialization and execution are intermixed, providing actual runtime information to the specializer. A performance evaluation concluded that the expected common speed-up is at least of 2 factors, up to 4 factors [83].

PyPy is an alternative implementation of the Python language that provides different optimizations to speed-up program execution [44]. PyPy is written in a subset of Python called RPython. RPython implements a tracing JIT compiler that uses runtime profile information to generate binary code of specialized versions of loops that take similar code paths [84]. PyPy has been evaluated to perform at least 351% faster than the rest of Python 2 implementations for common operations [45].

V8 is an open-source JavaScript engine included in the Chrome web browser and Node.js [46]. V8 implements a runtime adaptive JIT compiler that dynamically optimizes the generated code based on heuristics of the code execution profile. Since 2017, V8 includes the TurboFan JIT optimizing compiler that implements a multi-layered translation and an optimization pipeline to generate highly optimized machine code [85]. TurboFan specializes function contexts to optimize the execution of closures [86].

SpiderMonkey is the Mozilla's JavaScript engine, containing an interpreter and the IonMonkey JIT compiler [47]. IonMonkey includes many different optimizations such as function inlining, linear-scan register allocation, dead code elimination and loop-invariant code motion. IonMonkey also provides type specialization: during execution, the interpreter collects the actual types of variables; later, at JIT compilation, that type information is used to generate specialized code [87].

8. Conclusions

We have shown how compile-time program specialization is an appropriate mechanism to optimize gradually typed code. Moreover, the abstract interpretation performed by the program specializer can be used to provide static type safety. Every well-typed program can be specialized preserving its semantics, and the specialization process ensures termination. Possible threats to type safety are shown at compile time as warnings, and the execution of the specialized programs has been proved safe.

Our system performs many type checks statically that most gradually typed languages do at runtime. The outcome is that the specialized programs perform significantly better because fewer type checks are done dynamically. We have included our rule-based system in the *StaDyn* programming language, obtaining at least 93% average runtime performance benefit compared with the state-of-the-art optimizations for gradually typed code. When compared with the existing gradually typed languages for the same target platform (.NET), the specialized code is at least 13 times faster. This optimization requires no extra memory consumption at runtime, because program specialization takes place statically. We have measured the average compilation-time cost of program specialization to be 11.75%, when there are no type annotations.

We have not included dynamic code evaluation (e.g., the `eval` function available in Python, JavaScript and Lisp) in the core FC#^{G} language. In that case, program specialization could not be done statically, because the code to be evaluated is unknown at compile time. For this particular scenario, we plan to include the specializer in the runtime [6] in order to specialize the code just before its execution. A warning message will be shown to indicate potential type errors at runtime, but soundness will be guaranteed for the whole system. Aside from the cost of the dynamic program translation, the specialized code will benefit from the optimizations of code specialization.

The `dynamic` type is also used to interoperate with other dynamically typed programs written in other languages. In that case, no static type information can be inferred. As future work, we intend to work in combining our system with the classical gradual typing approach to provide the interoperation with other dynamically typed languages. For this kind of interaction, casts could be inserted to ensure type safety and increase interoperability, involving a decrease in runtime performance [69].

The source code and implementation of our program specializer and the *StaDyn* language, all the benchmarks and examples used in this article, and the evaluation data are freely available at http://www.reflection.uniovi.es/stadyn/download/2018/kbs

Acknowledgments

This work has been partially funded by the Spanish Department of Science, Innovation and Universities: project RTI2018-099235-B-I00. The authors have also received funds from the Banco Santander through its support to the Campus of International Excellence.

References

[1] J. G. Siek, M. M. Vitousek, M. Cimini, J. T. Boyland, Refined Criteria for Gradual Typing, in: LIPIcs - Leibniz International Proceedings in Informatics, Vol. 32, 2015, p. 293.

[2] J. G. Siek, W. Taha, Gradual Typing for Objects, in: Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP), Springer-Verlag, Berlin, Germany, 2007, pp. 2–27.

[3] E. Meijer, P. Drayton, Static Typing Where Possible Dynamic Typing When Needed: The End of the Cold War Between Programming Languages, in: Proceedings of the OOPSLA Workshop on Revival of Dynamic Languages, ACM, Vancouver, Canada, 2004, pp. 1–6.

[4] J. Strachan, Groovy 2.0 release notes, http://groovy-lang.org/releasenotes/groovy-2.0.html (2014).

[5] M. Zandstra, PHP Objects, Patterns, and Practice, 5th Edition, Apress, 2016.

[6] G. Bierman, E. Meijer, M. Torgersen, Adding Dynamic Types to C#, in: Proceedings of the 24th European Conference on Object-Oriented Programming, ECOOP, Springer-Verlag, Maribor, Slovenia, 2010, pp. 76–100.

[7] J. G. Siek, W. Taha, Gradual typing for functional languages, in: Scheme and Functional Programming Workshop, 2006, pp. 1–12.

[8] B. C. Pierce, Types and Programming Languages, The MIT Press, Cambridge, Massachusetts, 2002.

[9] L. Tratt, Chapter 5 Dynamically Typed Languages, in: Advances in Computers, Vol. 77, Elsevier Science Inc., 2009, Ch. 5, pp. 149–184.

[10] A. Takikawa, D. Feltey, B. Greenman, M. S. New, J. Vitek, M. Felleisen, Is Sound Gradual Typing Dead?, in: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, ACM, New York, NY, USA, 2016, pp. 456–468.

[11] L. P. Deutsch, A. M. Schiffman, Efficient implementation of the Smalltalk-80 system, in: Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages, POPL'84, ACM, New York, NY, USA, 1984, pp. 297–302.

[12] A. Kuhlenschmidt, D. Almahallawi, J. G. Siek, Towards Absolutely Efficient Gradually Typed Languages, in: European Conference on Object-Oriented Programming (ECOOP), Scripts to Programs Workshop (STOP), Prague, Czech Republic, 2015, pp. 1–2.

[13] S. Bauman, C. F. Bolz, R. Hirschfeld, V. Kirilichev, T. Pape, J. G. Siek, S. Tobin-Hochstadt, Pycket: a tracing JIT for a functional language, ACM SIGPLAN Notices 50 (9) (2015) 22–34.

[14] A. Takikawa, D. Feltey, E. Dean, M. Flatt, R. B. Findler, S. Tobin-hochstadt, Towards Practical Gradual Typing, in: European Conference on Object-Oriented Programming, Prague, Czech Republic, 2015, pp. 4–27.

[15] R. Chugh, P. M. Rondon, R. Jhala, Nested refinements: a logic for duck typing, in: Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL'12, ACM, New York, NY, USA, 2012, pp. 231–244.

[16] J. Quiroga, F. Ortin, D. Llewellyn-Jones, M. Garcia, Optimizing runtime performance of hybrid dynamically and statically typed languages for the .NET platform, Journal of Systems and Software 113.

[17] F. Ortin, J. Quiroga, J. M. Redondo, M. Garcia, Attaining Multiple Dispatch in Widespread Object-Oriented Languages, Dyna 182 (186) (2014) 242–250.

[18] L. Ina, I. Atsushi, Gradual Typing for Featherweight Java, Computer Software 26 (2) (2009) 18–40.

[19] A. Igarashi, B. C. Pierce, P. Wadler, Featherweight Java: a minimal core calculus for Java and GJ, ACM Transactions on Programming Languages and Systems 23 (3) (2001) 396–450.

[20] T. Zhao, J. Palsberg, J. Vitek, Lightweight confinement for featherweight java, in: Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications, OOPSLA '03, ACM, New York, NY, USA, 2003, pp. 135–148.

[21] S. Apel, C. Kästner, C. Lengauer, Feature featherweight java: A calculus for feature-oriented programming and stepwise refinement, in: Proceedings of the 7th International Conference on Generative Programming and Component Engineering, GPCE '08, ACM, New York, NY, USA, 2008, pp. 101–112.

[22] A. Igarashi, H. Nagira, Union types for object-oriented programming, in: Proceedings of the Symposium on Applied Computing (SAC), SAC '06, ACM, Dijon, France, 2006, pp. 1435–1441.

[23] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, D. Grossman, Enerj: Approximate data types for safe and general low-power computation, in: Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '11, ACM, New York, NY, USA, 2011, pp. 164–174.

[24] M. Dezani-Ciancaglini, D. Mostrous, N. Yoshida, S. Drossopoulou, Session types for object-oriented languages, in: D. Thomas (Ed.), ECOOP 2006 – Object-Oriented Programming, Springer Berlin Heidelberg, Berlin, Heidelberg, 2006, pp. 328–352.

[25] A. Igarashi, Formalizing Advanced Class Mechanisms, Ph.D. thesis, Tokyo (2000).

[26] B. C. Pierce, Programming with intersection types and bounded polymorphism, Tech. Rep. CMU-CS-91-106, School of Computer Science, Pittsburgh, PA, USA (1992).

[27] F. Barbanera, M. Dezani-Ciancaglini, U. De'Liguoro, Intersection and union types: syntax and semantics, Information and Computation 119 (2) (1995) 202–230.

[28] A. Aiken, E. L. Wimmers, Type Inclusion Constraints and Type Inference, in: Proceedings of the Conference on Functional Programming Languages and Computer Architecture, ACM, Copenhagen, Denmark, 1993, pp. 31–41.

[29] F. Ortin, M. Garcia, Union and intersection types to support both dynamic and static typing., Information Processing Letters 111 (6) (2011) 278–286.

[30] D. J. Pearce, Sound and complete flow typing with unions, intersections and negations, in: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), Vol. 7737 LNCS, 2013, pp. 335–354.

[31] F. Ortin, D. Zapico, J. B. G. Perez-Schofield, M. Garcia, Including both static and dynamic typing in the same programming language, IET Software 4 (4) (2010) 268–282.

[32] M. Garcia, F. Ortin, J. Quiroga, Design and implementation of an efficient hybrid dynamic and static typing language, Software: Practice and Experience 46 (2) (2015) 199–226.

[33] F. Ortin, The StaDyn programming language, http://www.reflection.uniovi.es/stadyn (2018).

[34] F. Ortin, Type Inference to Optimize a Hybrid Statically and Dynamically Typed Language, Computer Journal 54 (11) (2011) 1901–1924.

[35] B. Chiles, A. Turner, Dynamic Language Runtime, https://archive.codeplex.com/?p=dlr (2018).

[36] P. Vick, The Microsoft Visual Basic language specification, Microsoft Corporation, Redmond, Washington, 2007.

[37] R. B. De Oliveira, The Boo programming language, https://boo-language.github.io (2018).

[38] B. Frank, A. Frank, Fantom, the language formerly known as Fan, http://fantom.org (2018).

[39] J. Siegel, D. Frantz, H. Mirsky, R. Hudli, P. de Jong, A. Klein, B. Wilkins, A. Thomas, W. Coles, S. Baker, M. Balick, COBRA fundamentals and programming, 2nd Edition, John Wiley & Sons, Inc., New York, NY, USA, 2000.

[40] IronPython, The IronPython programming language, http://ironpython.net (2018).

[41] J. M. Redondo, F. Ortin, J. M. C. Lovelle, Optimizing Reflective Primitives of Dynamic Languages, International Journal of Software Engineering and Knowledge Engineering 18 (6) (2008) 759–783.

[42] F. Ortin, M. Garcia, StaDyn releases downloads, https://github.com/ComputationalReflection/StaDyn/releases (2018).

[43] J. Quiroga, F. Ortin, SSA transformations to facilitate type inference in dynamically typed code, The Computer Journal 60 (9) (2017) 1300–1315.

[44] A. Rigo, M. Fijalkowski, C. F. Bolz, A. Cuni, B. Peterson, A. Gaynor, PyPy, a fast, compliant alternative implementation of the Python Language, http://pypy.org (2018).

[45] J. M. Redondo, F. Ortin, A Comprehensive Evaluation of Widespread Python Implementations, IEEE Software 34 (4) (2015) 76–84.

[46] Google Inc., V8, the Google's high performance, open source, JavaScript engine, https://developers.google.com/v8 (2018).

[47] Mozilla, The SpiderMonkey JavaScript engine, https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey (2018).

[48] P. S. Foundation, Pybench benchmark project trunk page, http://svn.python.org/projects/python/trunk/Tools/pybench (2018).

[49] R. P. Weicker, Dhrystone: a synthetic systems programming benchmark, Communications of the ACM 27 (10) (1984) 1013–1030.

[50] C. A. Krintz, A Collection of Phoenix-Compatible C# Benchmarks, http://www.cs.ucsb.edu/ ckrintz/racelab/PhxCSBenchmarks (2018).

[51] A. Georges, D. Buytaert, L. Eeckhout, Statistically Rigorous Java Performance Evaluation, in: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications, OOPSLA, ACM, New York, NY, USA, 2007, pp. 57–76.

[52] D. J. Lilja, Measuring computer performance: a practitioner's guide, Cambridge University Press, 2005.

[53] Microsoft, The .NET compiler platform (Roslyn), http://msdn.microsoft.com/en-gb/roslyn (2018).

[54] Google, BigQuery, a fast, highly scalable, cost-effective, and fully managed cloud data warehouse for analytics, with built-in machine learning, https://cloud.google.com/bigquery (2019).

[55] R. Cartwright, M. Fagan, Soft typing, in: Proceedings of the Conference on Programming Language Design and Implementation (PLDI), ACM, Toronto, Canada, 1991, pp. 278–292.

[56] M. Abadi, L. Cardelli, B. C. Pierce, D. Rémy, Dynamic typing in polymorphic languages, Journal of Functional Programming 5 (1) (1995) 111–130.

[57] S. Thatte, Quasi-static typing, in: Proceedings of the 17th symposium on Principles of programming languages (POPL), ACM, San Francisco, California, United States, 1990, pp. 367–381.

[58] C. Flanagan, S. N. Freund, A. Tomb, Hybrid types, invariants, and refinements for imperative objects, in: Proceedings of the International Workshop on Foundations and Developments of Object-Oriented Languages (FOOL), ACM, San Antonio, Texas, 2006, pp. 1–11.

[59] I. Sergey, D. Clarke, Gradual ownership types, in: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), Vol. 7211 LNCS, 2012, pp. 579–599.

[60] K. A. Jafery, J. Dunfield, Sums of uncertainty: refinements go gradual, in: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, no. 1 in POPL, 2017, pp. 804–817.

[61] A. Igarashi, P. Thiemann, V. Vasconcelos, P. Wadler, Gradual session types, in: Proceedings of the ACM International Conference on Functional Programming, Vol. 1 of ICFP'17, Oxford, United Kingdom, 2017, pp. 1–38.

[62] R. Garcia, M. Cimini, Principal Type Schemes for Gradual Programs, in: Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '15, ACM, New York, NY, USA, 2015, pp. 303–315.

[63] G. Castagna, V. Lanvin, Gradual Typing with Union and Intersection Types, Proceedings of the ACM Programming Languages 1 (ICFP) (2017) 41:1–41:28.

[64] A. Rastogi, N. Swamy, C. Fournet, G. Bierman, P. Vekris, Safe & Efficient gradual typing for type script, in: Conference Record of the Annual ACM Symposium on Principles of Programming Languages, Vol. 2015-Janua, 2015, pp. 167–180.

[65] J. G. Siek, M. M. Vitousek, M. Cimini, S. Tobin-Hochstadt, R. Garcia, Monotonic References for Efficient Gradual Typing, in: J. Vitek (Ed.), Programming Languages and Systems, Springer Berlin Heidelberg, Berlin, Heidelberg, 2015, pp. 432–456.

[66] D. Herman, A. Tomb, C. Flanagan, Space-efficient gradual typing, Higher-Order and Symbolic Computation 23 (2) (2010) 167–189.

[67] D. Ancona, M. Ancona, A. Cuni, N. D. Matsakis, RPython: a step towards reconciling dynamically and statically typed OO languages, Proceedings of the 2007 symposium on Dynamic languages (2007) 53–64.

[68] S. Bauman, C. Friedrich Bolz-Tereick, J. Siek, S. Tobin, Sound Gradual Typing: Only Mostly Dead, Proceedings of the ACM on Object-Oriented Programming, Systems, Languages & Applications 1 (OOPSLA) (2017) 1–24.

[69] J. G. Siek, Challenges and progress toward efficient gradual typing, in: Dynamic Languages Symposium, 2017, pp. 1–2.

[70] J. P. Campora, S. Chen, M. Erwig, E. Walkingshaw, Migrating Gradual Types, in: Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages, POPL, ACM, New York, NY, USA, 2018, pp. 15:1–15:29.

[71] Google Open Source, Pytype, a static type inferencer for Python code, https://opensource.google.com/projects/pytype (2018).

[72] G. van Rossum, J. Lehtosalo, L. Langa, Python Enhanced Proposal (PEP) 484. Type hints, https://www.python.org/dev/peps/pep-0484 (2014).

[73] M. Kramm, Pytype source code at GitHub, https://github.com/google/pytype (2018).

[74] M. Kramm, Python typology, in: PyCon Conferencence, PyCon, 2016, pp. 804–817.

[75] F. Ortin, J. M. Redondo, J. B. G. Perez-Schofield, Towards a Static Type Checker for Python, in: European Conference on Object-Oriented Programming (ECOOP), Scripts to Programs Workshop (STOP), STOP'15, Prague, Czech Republic, 2015, pp. 1–4.

[76] J. M. Redondo, F. Ortin, *stypy*: a static type checker for the Python language, https://github.com/computationalReflection/stypy (2018).

[77] V. Raychev, M. Vechev, A. Krause, Predicting Program Properties from "Big Code", in: Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '15, ACM, New York, NY, USA, 2015, pp. 111–124.

[78] N. D. Jones, C. K. Gomard, P. Sestoft, Partial Evaluation and Automatic Program Generation, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[79] Y. Futamura, Partial evaluation of computation process–an approach to a compiler-compiler, Higher-Order and Symbolic Computation 12 (4) (1999) 381–391.

[80] U. P. Schultz, Partial evaluation for class-based object-oriented languages, in: Proceedings of the Second Symposium on Programs As Data Objects, PADO '01, Springer-Verlag, London, UK, UK, 2001, pp. 173–197.

[81] U. P. Schultz, Object-Oriented Software Engineering Using Partial Evaluation, Ph.D. thesis, University of Rennes (2000).

[82] A. Rigo, Representation-based just-in-time specialization and the psyco prototype for python, PEPM '04: Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation (2004) 15–26.

[83] A. Rigo, Psyco, high-level languages do not need to be slower than low-level ones, http://psyco.sourceforge.net (2012).

[84] C. F. Bolz, A. Cuni, M. Fijalkowski, A. Rigo, Tracing the meta-level: PyPy's tracing JIT compiler, in: Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems, ICOOOLPS'09, ACM, New York, NY, USA, 2009, pp. 18–25.

[85] TurboFan, V8's optimizing compilers leveraging a concept called Sea of Nodes, https://github.com/v8/v8/wiki/TurboFan (2018).

[86] J. Sevcik, Function context specialization, https://docs.google.com/document/d/-1CJbBtqzKmQxM1Mo4xU0ENA7KXqb1YzI6HQU8qESZ9Ic (2016).

[87] Mozilla, IonMonkey Middle-level Intermediate Representation (MIR), https://wiki.mozilla.org/IonMonkey/MIR (2018).

Appendix A. Auxiliary functions used in FC#

$$\frac{\texttt{class}\, C : C_{super}\, \texttt{\{}\, \overline{T_f\, f}\,\texttt{;}\; K\, \overline{M}\, \texttt{\}} \in \overline{CD} \qquad fields(\overline{CD}, C_{super}) = \overline{T_{super}\, f_{super}}}{fields(\overline{CD}, C) = \overline{T_f\, f}, \overline{T_{super}\, f_{super}}} \qquad \frac{}{fields(\overline{CD}, \texttt{object}) = \varnothing}$$

$$\frac{\overline{CD} = CD_1 \ldots \texttt{class}\, C : C_{super}\, \texttt{\{}\, \overline{T_f\, f}\,\texttt{;}\; K\, \overline{M}\, \texttt{\}} \ldots CD_n \qquad \overline{M} = M_1 \ldots T_r\, m(\overline{T_p\, x})\, \texttt{\{}\, \texttt{return}\, e\texttt{;}\texttt{\}} \ldots M_m}{method(\overline{CD}, C, m) = T_r\, m(\overline{T_p\, x})\, \texttt{\{}\, \texttt{return}\, e\texttt{;}\texttt{\}}}$$

$$\frac{\overline{CD} = CD_1 \ldots \texttt{class}\, C : C_{super}\, \texttt{\{}\, \overline{T_f\, f}\,\texttt{;}\; K\, \overline{M}\, \texttt{\}} \ldots CD_n \qquad m \notin \overline{M}}{method(\overline{CD}, C, m) = method(\overline{CD}, C_{super}, m)}$$

$$\frac{}{method(\overline{CD}, \texttt{object}, m) = \varnothing} \qquad \frac{\overline{M} = M_1 \ldots T_r\, m(\overline{T_p\, x})\, \texttt{\{}\, \texttt{return}\, e\texttt{;}\texttt{\}} \ldots M_m}{type(M) = \overline{T_p} \to T_r}$$

Figure A.1: *fields*, *method* and *type* auxiliary functions used in FC#.

Appendix B. Well-formedness of FC#$^{\mathrm{G}}$

Figure B.1 shows the well-formedness rules for FC#$^{\mathrm{G}}$. W-Prog$^{\mathrm{G}}$ checks well formedness of a whole program by checking each class definition and the main expression. Therefore, a program $P^G = \overline{CD^G}\, e$ will be accepted by the type system when $\overline{CD^G}, \varnothing \vdash e :^G \diamond$ is derived from the existing rules.

For each class definition (W-Class$^{\mathrm{G}}$), class identifiers must be unique, base classes must also be defined, there could not be cycles in the subtyping relation and, like in FJ, inherited fields cannot be redefined in subclasses.

Unlike FJ, FC#$^{\mathrm{G}}$ allows defining different types for fields and the corresponding constructor parameters. Therefore, W-Const$^{\mathrm{G}}$ checks that the constructor parameters are consistent subtypes ($\lesssim$) of the field types. The base constructor invocation is also type checked.

W-Method$^{\mathrm{G}}$ requires method bodies to return a consistent subtype of the declared return type. In addition, derived methods must have the same exact signature as the method with the same name in the base class (if any), transitively (i.e., method overriding).

(W-Prog$^{\mathrm{G}}$)

$$\frac{\forall\, CD_i^G \in \overline{CD^G}\,.\,\overline{CD^G} - \{CD_i^G\}, \Gamma \vdash CD_i^G :^G \diamond \qquad \overline{CD^G}, \Gamma \vdash e :^G T^G}{\varnothing, \Gamma \vdash \overline{CD^G}\, e :^G \diamond}$$

(W-Class$^{\mathrm{G}}$)

$$\frac{\begin{array}{c} C \notin \overline{CD^G} \qquad C_{super} \in \overline{CD^G} \\ \overline{CD^G} \vdash C_{super} \not\leq C \qquad \overline{CD^G}, \Gamma \vdash K :^G \diamond \qquad fields(\overline{CD^G}, C_{super}) = \overline{T^G_{super_f}\, f_{super}} \\ \overline{f} \cap \overline{f_{super}} = \varnothing \qquad \forall\, M_i^G \in \overline{M^G}\,.\,\overline{CD^G}, \Gamma \vdash M_i^G :^G_C \diamond \end{array}}{\overline{CD^G}, \Gamma \vdash \texttt{class}\, C\texttt{:}\, C_{super}\,\texttt{\{}\,\overline{T_f^G\, f}\,\texttt{;}\ K^G\, \overline{M^G}\,\texttt{\}} :^G \diamond}$$

(W-Const$^{\mathrm{G}}$)

$$\frac{\begin{array}{c} \overline{CD^G} = CD_1^G \ldots \texttt{class}\, C : C_{super}\,\texttt{\{}\,\overline{T^G_{f2}\, f_2}\,\texttt{;}\ K^G\, \overline{M^G}\,\texttt{\}} \ldots CD_n^G \\ \overline{CD^G} \vdash \overline{T^G_{p2}} \lesssim \overline{T^G_{f2}} \qquad \Gamma' = \overline{x_{arg1}} : \overline{T^G_{p1}},\, \Gamma \qquad \overline{CD^G}, \Gamma' \vdash \texttt{new}\, C_{super}(\overline{x_{arg1}}) :^G C_{super} \end{array}}{\overline{CD^G}, \Gamma \vdash C(\overline{T^G_{p1}\, f_1}, \overline{T^G_{p2}\, f_2})\texttt{: base(}\overline{f_1}\texttt{)}\ \texttt{\{}\,\overline{\texttt{this.}f_2 = f_2\texttt{;}}\,\texttt{\}} :^G \diamond}$$

(W-Method$^{\mathrm{G}}$)

$$\frac{\begin{array}{c} \Gamma' = x : \overline{T_p^G},\, \texttt{this} : C,\, \Gamma \\ \overline{CD^G}, \Gamma' \vdash e :^G T^G_{body} \qquad \texttt{class}\, C : C_{super}\,\texttt{\{}\,\overline{T_f^G\, f}\,\texttt{;}\ K^G\, \overline{M^G}\,\texttt{\}} \in \overline{CD^G} \\ T_r^G\, m(\overline{T_p^G\, x})\,\texttt{\{return}\, e\texttt{;\}} \in \overline{M^G} \qquad T^G_{body} \lesssim T_r^G \\ \text{if } type(method(\overline{CD^G}, C_{super}, m)) = \overline{T^G_{super_p}} \to T^G_{super_r} \text{ then } \overline{T^G_{super_p}} = \overline{T_p^G} \text{ and } T^G_{super_r} = T_r^G \end{array}}{\overline{CD^G}, \Gamma \vdash T_r^G\, m(\overline{T_p^G\, x})\,\texttt{\{return}\, e\ \texttt{;\}} :^G_C \diamond}$$

Figure B.1: Well-formedness rules for FC#$^{\mathrm{G}}$.

Appendix C. Congruence rules for the semantics of FC#$^{\mathrm{G}}$

(R-FieldC$^{\mathrm{G}}$)
$$\frac{\overline{CD^G} \vdash e \longrightarrow^G e'}{\overline{CD^G} \vdash e.f \longrightarrow^G e'.f}$$

(R-InvC$^{\mathrm{G}}$)
$$\frac{\overline{CD^G} \vdash e \longrightarrow^G e'}{\overline{CD^G} \vdash e.m(\overline{e_{arg}}) \longrightarrow^G e'.m(\overline{e_{arg}})}$$

(R-InvAC$^{\mathrm{G}}$)
$$\frac{\overline{CD^G} \vdash e \longrightarrow^G e'}{\overline{CD^G} \vdash \sigma.m(\overline{\sigma}, e, \ldots) \longrightarrow^G \sigma.m(\overline{\sigma}, e', \ldots)}$$

(R-NewAC$^{\mathrm{G}}$)
$$\frac{\overline{CD^G} \vdash e \longrightarrow^G e'}{\overline{CD^G} \vdash \texttt{new}\, C(\overline{\sigma}, e, \ldots) \longrightarrow^G \texttt{new}\, C(\overline{\sigma}, e', \ldots)}$$

Figure C.1: Congruence semantic rules of FC#$^{\mathrm{G}}$.

Appendix D. Progress of FC#$^{\mathrm{G}}$

Lemma 1. *Type of values are not dynamic in FC#^{G}.*

If $\vdash P^G = \overline{CD^G}\, e_1 :^G \diamond$ and $\overline{CD^G}, \varnothing \vdash e_1 :^G dynamic$, then $e_1 \notin Values$.

Proof. By induction on the typing derivation.

The only rule typing an expression that follows the syntax of values is T-New$^{\mathrm{G}}$, and its conclusion types the expression as a class. □

Property 2. *Progress of FC#^{G}.*

If $\vdash P^G = \overline{CD^G}\, e_1 :^G \diamond$ and $\overline{CD^G}, \varnothing \vdash e_1 :^G T_1^G$, then

- $e_1 \in$ Values, or
- $\overline{CD^G} \vdash e_1 \longrightarrow^G error$, or
- $\exists\, e_2 \,.\, \overline{CD^G} \vdash e_1 \longrightarrow^G e_2$

Proof. By induction on the typing derivation.

- Cases T-Var$^{\mathrm{G}}$ and T-This$^{\mathrm{G}}$. These cases cannot occur because Γ in the hypothesis is empty (i.e., the expression e is closed). Thus, the premises in T-Var$^{\mathrm{G}}$ and T-This$^{\mathrm{G}}$ are not fulfilled.
- Case T-FieldC$^{\mathrm{G}}$. $\quad e_1 = e.f \qquad T_1^G = T^G \qquad \overline{CD^G}, \varnothing \vdash e :^G C$

  $fields(\overline{CD^G}, C) = T_1^G\, f_1 \ldots T^G\, f \ldots T_n^G\, f_n$

  e_1 may be a value of another expression. For the first case, C, the type of e, has the requested f field, so a reduction could be done by R-Var$^{\mathrm{G}}$. Otherwise, R-FieldC$^{\mathrm{G}}$ is reduced.
- Case T-FieldD$^{\mathrm{G}}$. $\quad e_1 = e.f \qquad T_1^G = dynamic \qquad \overline{CD^G}, \varnothing \vdash e :^G dynamic$

  By Lemma 1, e_1 is not a value. Then, R-FieldC$^{\mathrm{G}}$ is reduced by the induction hypothesis.

– Case T-InvC$^{\text{G}}$. $e_1 = e_{obj}.m(\overline{e_{arg}})$ $\quad T_1^G = T_r^G$ $\quad \overline{CD^G}, \Gamma \vdash e_{obj} :^G C_{obj}$
$\mathit{type}(\mathit{method}(\overline{CD^G}, C_{obj}, m)) = \overline{T_p^G} \to T_r^G$ $\quad \overline{CD_G}, \Gamma \vdash \overline{e_{arg}} :^G \overline{T_{arg}^G}$
$\overline{CD^G} \vdash \overline{T_{arg}^G} \lesssim T_p^G$

Since C_{obj} has the m method and the arguments must be consistent subtypes of the parameter types, R-Inv$^{\text{G}}$ can be reduced.

– Case T-InvD$^{\text{G}}$. $e_1 = e_{obj}.m(\overline{e_{arg}})$ $\quad T_1^G = \mathit{dynamic}$ $\quad \overline{CD^G}, \Gamma \vdash e_{obj} :^G \mathit{dynamic}$
$\overline{CD^G}, \Gamma \vdash \overline{e_{arg}} :^G \overline{T_{arg}^G}$

As the type of e_{obj} is `dynamic`, it is not a value (Lemma 1), so either R-InvC$^{\text{G}}$ or R-InvAC$^{\text{G}}$ is reduced.

□

Appendix E. Abstract syntax of FC#$^{\text{S}}$

P^S	$\in$	Program	::=	$\overline{CD^S}\ e$
CD^S	$\in$	Class definition	::=	`class` C `:` C `{` $\overline{T^S\, f}$`;` K^S $\overline{M^S}$ `}`
K^S	$\in$	Constructor	::=	C`(`$\overline{T^S\, f}$`):base(`$\overline{f}$`) {` $\overline{\texttt{this.}f\texttt{=}f}$`; }`
M^S	$\in$	Method	::=	$T^S\, m(\overline{T^S\, x})$ `{ return` e`; }`
e	$\in$	Expression	::=	x | `this` | $e.f$ | $e.m(\overline{e})$ | `new` $C(\overline{e})$
T^S	$\in$	Type	::=	$T^S \vee T^S$ | C

Figure E.1: Syntax of FC#$^{\text{S}}$.

Appendix F. Type system of FC#$^{\mathbf{S}}$

$$\textsc{(W-Prog}^{\textsc{S}}\textsc{)}\quad \frac{\forall\, CD_i^S \in \overline{CD^S}\,.\,\overline{CD^S} - \{CD_i^S\}, \Gamma \vdash CD_i^S :^S \diamond \qquad \overline{CD^S}, \Gamma \vdash e :^S T^S}{\varnothing, \Gamma \vdash \overline{CD^S}\, e :^S \diamond}$$

$$\textsc{(W-Class}^{\textsc{S}}\textsc{)}\quad \frac{\begin{array}{c} C \notin \overline{CD^S} \qquad C_{super} \in \overline{CD^S} \qquad \overline{CD^S} \vdash C_{super} \not\leq C \qquad \overline{CD^S}, \Gamma \vdash K :^S \diamond \\ fields(\overline{CD^S}, C_{super}) = \overline{T^S_{superf} f_{super}} \qquad \overline{f} \cap \overline{f_{super}} = \varnothing \qquad \forall\, M_i^S \in \overline{M^S}\,.\,\overline{CD^S}, \Gamma \vdash M_i^S :^S_C \diamond \end{array}}{\overline{CD^S}, \Gamma \vdash \texttt{class}\, C \texttt{:}\, C_{super}\, \texttt{\{}\, \overline{T_f^S f}\, \texttt{;}\, K^S\, \overline{M^S}\, \texttt{\}} :^S \diamond}$$

$$\textsc{(W-Const}^{\textsc{S}}\textsc{)}\quad \frac{\begin{array}{c} \overline{CD^S} = CD_1^S \ldots \texttt{class}\, C : C_{super}\, \texttt{\{}\, \overline{T^S_{f2} f2}\texttt{;}\, K^S\, \overline{M^S}\, \texttt{\}} \ldots CD_n^S \\ \overline{CD^S} \vdash \overline{T^S_{p2}} < \overline{T^S_{f2}} \qquad \Gamma' = \overline{x_{arg1}} : \overline{T^S_{p1}}, \Gamma \qquad \overline{CD^S}, \Gamma' \vdash \texttt{new}\, C_{super}(\overline{x_{arg1}}) :^S C_{super} \end{array}}{\overline{CD^S}, \Gamma \vdash C(\overline{T^S_{p1}\, f_1}, \overline{T^S_{p2}\, f_2})\texttt{: base(}\overline{f_1}\texttt{)}\, \texttt{\{}\, \overline{\texttt{this.}f_2 = f_2}\texttt{;\}} :^S \diamond}$$

$$\textsc{(W-Method}^{\textsc{S}}\textsc{)}\quad \frac{\begin{array}{c} \Gamma' = \overline{x} : \overline{T_p^S}, \texttt{this} : C, \Gamma \\ \overline{CD^S}, \Gamma' \vdash e :^S T^S_{body} \qquad \texttt{class}\, C : C_{super}\, \texttt{\{}\, \overline{T_f^S f}\, \texttt{;}\, K^S\, \overline{M^S}\, \texttt{\}} \in \overline{CD^S} \\ T_r^S\, m(\overline{T_p^S\, x})\, \texttt{\{return}\, e\, \texttt{;\}} \in \overline{M^S} \qquad T^S_{body} \leq T_r^S \\ \text{if } type(method(\overline{CD^S}, C_{super}, m)) = \overline{T^S_{superp}} \to T^S_{superr} \text{ then } \overline{T^S_{superp}} = \overline{T_p^S} \text{ and } T^S_{superr} = T_r^S \end{array}}{\overline{CD^S}, \Gamma \vdash T_r^S\, m(\overline{T_p^S\, x})\, \texttt{\{return}\, e\, \texttt{;\}} :^S_C \diamond}$$

Figure F.1: Well-formedness rules for FC#$^{\mathrm{S}}$.

(T-Var$^{\mathrm{S}}$)

$$\frac{}{\overline{CD^S}, \Gamma \vdash x :^S \Gamma(x)}$$

(T-This$^{\mathrm{S}}$)

$$\frac{}{\overline{CD^S}, \Gamma \vdash \texttt{this} :^S \Gamma(\texttt{this})}$$

(T-FieldC$^{\mathrm{S}}$)

$$\frac{\overline{CD^S}, \Gamma \vdash e :^S C \qquad \mathit{fields}(\overline{CD^S}, C) = T^S_1 f_1 \ldots T^S f \ldots T^S_n f_n}{\overline{CD^S}, \Gamma \vdash e.f :^S T^S}$$

(T-FieldU$^{\mathrm{S}}$)

$$\frac{\begin{array}{c} \overline{CD^S}, \Gamma \vdash e :^S C_1 \vee \ldots \vee C_n \qquad \overline{C_{withf}} = \{C_i \in C_1, \ldots, C_n\} \,.\, T_{f_i} f \in \mathit{fields}(\overline{CD^S}, C_i)\} \\ n_f = \#(\overline{C_{withf}}) \geq 1 \qquad \overline{C_{wrong}} = \{C_1, \ldots, C_n\} - \overline{C_{withf}} \\ \text{if } \#(\overline{C_{wrong}}) \geq 1 \text{ then } \mathit{warning}(\text{“Field } f \text{ not found in types } \overline{C_{wrong}}\text{”}) \end{array}}{\overline{CD^S}, \Gamma \vdash e.f :^S T_{f_1} \vee \ldots \vee T_{f_{n_f}}}$$

(T-InvC$^{\mathrm{S}}$)

$$\frac{\begin{array}{c} \overline{CD^S}, \Gamma \vdash e_{obj} :^S C_{obj} \\ \mathit{type}(\mathit{method}(\overline{CD^S}, C_{obj}, m)) = \overline{T^S_p} \rightarrow T^S_r \qquad \overline{CD^S}, \Gamma \vdash \overline{e_{arg}} :^S \overline{T^S_{arg}} \qquad \overline{CD^S} \vdash \overline{T^S_{arg}} \leq \overline{T^S_p} \end{array}}{\overline{CD^S}, \Gamma \vdash e_{obj}.m(\overline{e_{arg}}) :^S T^S_r}$$

(T-InvU$^{\mathrm{S}}$)

$$\frac{\begin{array}{c} \overline{CD^S}, \Gamma \vdash e :^S C_1 \vee \ldots \vee C_n \qquad \overline{CD^S}, \Gamma \vdash \overline{e_{arg}} :^S \overline{T^S_{arg}} \\ \overline{C_{withm}} = \{C_i \in C_1, \ldots, C_n\} \,.\, \mathit{type}(\mathit{method}(\overline{CD^S}, C_i, m)) = \overline{T^S_{p_i}} \rightarrow T^S_{r_i} \text{ and } \overline{T^S_{arg}} \leq \overline{T^S_{p_i}} \\ n_m = \#(\overline{C_{withm}}) \geq 1 \qquad \overline{C_{wrong}} = \{C_1, \ldots, C_n\} - \overline{C_{withm}} \\ \text{if } \#(\overline{C_{wrong}}) \geq 1 \text{ then } \mathit{warning}(\text{“The classes } \overline{C_{wrong}} \text{ do not provide a suitable } m \text{ method”}) \end{array}}{\overline{CD^S}, \Gamma \vdash e.m(\overline{e_{arg}}) :^S T_{r_1} \vee \ldots \vee T_{r_{n_m}}}$$

(T-New$^{\mathrm{S}}$)

$$\frac{\begin{array}{c} \overline{CD^S} = CD^S_1 \ldots \texttt{class}\, C : C_{super}\, \texttt{\{}\, \overline{T^S_{f2} f_2}\,\texttt{;}\; K\; \overline{M}\; \texttt{\}} \ldots CD^S_n \\ K^S = C(\overline{T^S_{p1}\, f_1}, \overline{T^S_{p2}\, f_2})\texttt{: base(}\overline{f_1}\texttt{) \{}\overline{\texttt{this.}f_2 = f_2\texttt{;}}\texttt{\}} \qquad \overline{CD^S}, \Gamma \vdash \overline{e_{arg1}} :^S \overline{T^S_{arg1}} \\ \overline{CD^S} \vdash \overline{T^S_{arg1}} \leq \overline{T^S_{p1}} \qquad \overline{CD^S}, \Gamma \vdash \overline{e_{arg2}} :^S \overline{T^S_{arg2}} \qquad \overline{CD^S} \vdash \overline{T^S_{arg2}} \leq \overline{T^S_{p2}} \end{array}}{\overline{CD^S}, \Gamma \vdash \texttt{new}\, C(\overline{e_{arg1}}, \overline{e_{arg2}}) :^S C}$$

Figure F.2: Type system of FC#$^{\mathrm{S}}$.

Appendix G. Semantics of FC#$^{\text{S}}$

(R-Field$^{\text{S}}$)
$$\frac{\mathit{fields}(\overline{CD^S}, C) = T_1^S f_1 \ldots T_i^S f \ldots T_n^S f_n \qquad \overline{\sigma} = \sigma_1 \ldots \sigma_i \ldots \sigma_n}{\overline{CD^S} \vdash \texttt{new}\, C(\overline{\sigma}).f \longrightarrow^S \sigma_i}$$

(R-FieldC$^{\text{S}}$)
$$\frac{\overline{CD^S} \vdash e \longrightarrow^S e'}{\overline{CD^S} \vdash e.f \longrightarrow^S e'.f}$$

(R-Inv$^{\text{S}}$)
$$\frac{\mathit{method}(\overline{CD^S}, C_{obj}, m) = T_r^S\, m(\overline{T_p^S\, x})\,\texttt{\{}\,\texttt{return}\, e_{body}\,\texttt{;\}} \qquad \overline{CD^S} \vdash \overline{C_{arg}} \leq \overline{T_p}}{\begin{array}{l}\overline{CD^S} \vdash \texttt{new}\ C_{obj}(\overline{\sigma_{obj}}).m(\texttt{new}\ C_{arg_1}(\overline{\sigma_{arg_1}}), \ldots, \texttt{new}\ C_{arg_n}(\overline{\sigma_{arg_n}})) \longrightarrow^S \\ \quad [\texttt{new}\ C_{arg_1}(\overline{\sigma_{arg_1}})/x_1, \ldots, \texttt{new}\ C_{arg_n}(\overline{\sigma_{arg_n}})/x_n, \texttt{new}\ C_{obj}(\overline{\sigma_{obj}})/\texttt{this}]e_{body}\end{array}}$$

(R-InvC$^{\text{S}}$)
$$\frac{\overline{CD^S} \vdash e \longrightarrow^S e'}{\overline{CD^S} \vdash e.m(\overline{e_{arg}}) \longrightarrow^S e'.m(\overline{e_{arg}})}$$

(R-InvAC$^{\text{S}}$)
$$\frac{\overline{CD^S} \vdash e \longrightarrow^S e'}{\overline{CD^S} \vdash \sigma.m(\overline{\sigma}, e, \ldots) \longrightarrow^S \sigma.m(\overline{\sigma}, e', \ldots)}$$

(R-NewAC$^{\text{S}}$)
$$\frac{\overline{CD^S} \vdash e \longrightarrow^S e'}{\overline{CD^S} \vdash \texttt{new}\, C(\overline{\sigma}, e, \ldots) \longrightarrow^S \texttt{new}\, C(\overline{\sigma}, e', \ldots)}$$

(R-Err$^{\text{S}}$)
$$\frac{\overline{CD^S} \vdash e \longrightarrow^S e'}{\overline{CD^S} \vdash e \longrightarrow^{SE} e'}$$

(R-FieldE$^{\text{S}}$)
$$\frac{\mathit{fields}(\overline{CD^S}, C) = \overline{T^S f} \qquad f \notin \overline{f}}{\overline{CD^S} \vdash \texttt{new}\, C(\overline{\sigma}).f \longrightarrow^{SE} \mathit{error}(\text{“Field } f \text{ not found”})}$$

(R-InvE$^{\text{S}}$)
$$\frac{\mathit{method}(\overline{CD^S}, C, m) = \varnothing}{\overline{CD^S} \vdash \texttt{new}\, C(\overline{\sigma_{obj}}).m(\overline{\sigma_{arg}}) \longrightarrow^{SE} \mathit{error}(\text{“Method } m \text{ not found”})}$$

(R-ParE$^{\text{S}}$)
$$\frac{\mathit{method}(\overline{CD^S}, C, m) = T_r^S\, m(\overline{T_p^S\, x})\,\texttt{\{}\,\texttt{return}\, e\texttt{;\}} \qquad \#(\overline{\sigma_{arg}}) \neq \#(\overline{x})}{\overline{CD^S} \vdash \texttt{new}\, C(\overline{\sigma_{obj}}).m(\overline{\sigma_{arg}}) \longrightarrow^{SE} \mathit{error}(\text{“Wrong number of arguments”})}$$

(R-ArgE$^{\text{S}}$)
$$\frac{\begin{array}{c}\mathit{method}(\overline{CD^S}, C, m) = T_r^S\, m(\overline{T_p^S\, x})\,\texttt{\{}\,\texttt{return}\, e\texttt{;\}} \\ \#(\overline{x}) = n \qquad \exists\, i \in [1, n]\,.\,\overline{CD^S} \vdash C_{arg_i} \nleq T_{p_i}\end{array}}{\begin{array}{c}\overline{CD^S} \vdash \texttt{new}\ C_{obj}(\overline{\sigma_{obj}}).m(\texttt{new}\ C_{arg_1}(\overline{\sigma_{arg_1}}), \ldots, \texttt{new}\ C_{arg_n}(\overline{\sigma_{arg_n}})) \longrightarrow^{SE} \\ \mathit{error}(\text{“Wrong type of the } i^{th}\text{argument”})\end{array}}$$

Figure G.1: Semantics of FC#$^{\text{S}}$.

Appendix H. Progress of FC#$^{\mathrm{S}}$

Lemma 2. *Values are not union types in FC#^{S}.*

If $\vdash P^S = \overline{CD^S}\, e_1 :^S \diamond$ and $\overline{CD^S}, \varnothing \vdash e_1 :^S T_1 \vee \ldots \vee T_n$, then $e_1 \notin Values$.

Proof. By induction on the typing derivation.

The only rule that types an expression following the syntax of a value is T-New$^{\mathrm{G}}$, and its conclusion types the expression as a class. □

Property 3. *Progress of FC#^{S}.*

If $\vdash P^S = \overline{CD^S}\, e_1 :^S \diamond$ and $\overline{CD^S}, \varnothing \vdash e_1 :^S T_1^S$, then

– $e_1 \in$ Values, or
– $\overline{CD^S} \vdash e_1 \longrightarrow^{SE} error$, or
– $\exists\, e_2 \,.\, \overline{CD^S} \vdash e_1 \longrightarrow^{SE} e_2$

Proof. By induction on the typing derivation.

– Cases T-Var$^{\mathrm{S}}$, T-This$^{\mathrm{S}}$, T-FieldC$^{\mathrm{S}}$ and T-InvC$^{\mathrm{S}}$ are proved the same as in Appendix D.

– Case T-FieldU$^{\mathrm{S}}$. $\quad e_1 = e.f \qquad T_1^S = T_{f_1}^S \vee \ldots \vee T_{f_{n_f}}^S \qquad \overline{CD^S}, \varnothing \vdash e :^S C_1 \vee \ldots \vee C_n$

  Lemma 2 ensures that e is not a value, because its type is a union type. Therefore, R-FieldC$^{\mathrm{S}}$ is reduced by the induction hypothesis.

– Case T-InvU$^{\mathrm{S}}$. $\quad e_1 = e.m(\overline{e_{arg}}) \qquad T_1^S = T_{r_1}^S \vee \ldots \vee T_{r_{n_m}}^S \qquad \overline{CD^S}, \varnothing \vdash e :^S C_1 \vee \ldots \vee C_n$

  Similar to the previous case. By Lemma 2, e is not a value. Thus, the induction hypothesis tells us that R-InvC$^{\mathrm{S}}$ is reduced.

□

Appendix I. Static type safety of FC#$^{\mathrm{S}}$ without warnings

Lemma 3. *Subtyping of union types without warnings.*

The S-UnionL$^{\mathrm{S}}$ rule, when warnings are considered as type errors, is equivalent to:

$$\frac{\forall\, C_i \in C_1, \ldots, C_n \,.\, \overline{CD^S} \vdash C_i \leq C}{\overline{CD^S} \vdash C_1 \vee \ldots \vee C_n \leq C}$$

Proof. Since there cannot be any warnings in the premises, $\{C_1, \ldots, C_n\} - \overline{C_{sub}} = \varnothing$ (see the S-UnionL$^{\mathrm{S}}$ original rule). Then, $\overline{C_{sub}} = \{C_1, \ldots, C_n\}$, so the $\overline{C_{sub}}$ set is made up of all the types in the union type. □

Lemma 4. *Field access of union types without warnings.*

T-FieldU$^{\mathrm{S}}$, when warnings are considered as type errors, is equivalent to:

$$\frac{\overline{CD^S}, \Gamma \vdash e :^S C_1 \vee \ldots \vee C_n \qquad \forall\, C_i \in C_1, \ldots, C_n \,.\, T^S_{f_i} f \in \mathit{fields}(\overline{CD^S}, C_i)}{\overline{CD^S}, \Gamma \vdash e.f :^S T^S_{f_1} \vee \ldots \vee T_{f_n}}$$

Proof. There cannot be any warnings in the premises, so $\{C_1, \ldots, C_n\} - \overline{C_{withf}} = \varnothing$ (see the T-FieldU$^{\mathrm{S}}$ original rule). Then, $\overline{C_{withf}} = \{C_1, \ldots, C_n\}$, so $\overline{C_{withf}}$ is made up of all the types in the union type. □

Lemma 5. *Invocation of union types without warnings.*

T-InvU$^{\mathrm{S}}$, when warnings are considered as type errors, is equivalent to:

$$\frac{\begin{array}{c}\overline{CD^S}, \Gamma \vdash e :^S C_1 \vee \ldots \vee C_n \qquad \overline{CD^S}, \Gamma \vdash \overline{e_{arg}} :^S \overline{T^S_{arg}} \\ \forall\, C_i \in C_1, \ldots, C_n \,.\, \mathit{type}(\mathit{method}(\overline{CD^S}, C_i, m)) = \overline{T^S_{p_i}} \to T^S_{r_i} \text{ and } \overline{T^S_{arg}} \leq \overline{T^S_{p_i}}\end{array}}{\overline{CD^S}, \Gamma \vdash e.m(\overline{e_{arg}}) :^S T^S_{r_1} \vee \ldots \vee T^S_{r_n}}$$

Proof. Similar to Lemma 4. □

Lemma 6. *Argument substitution preserves typing in FC#^{S}.*

If $\vdash P^S = \overline{CD^S}\, e_1 :^S \diamond$ and $\overline{CD^S}, \Gamma, \overline{x}{:}\overline{T^S_x} \vdash e_1 :^S T^S_1$ and $\overline{CD^S}, \Gamma \vdash \overline{e_2} :^S \overline{T^S_2}$ and $\overline{CD^S} \vdash \overline{T^S_2} \leq \overline{T^S_x}$, then $\overline{CD^S}, \Gamma \vdash [\overline{e_2}/\overline{x}]e_1 :^S T^S_3$ for some $\overline{CD^S} \vdash T^S_3 \leq T^S_1$.

Proof. By induction on the typing derivation.

- Case T-Var$^{\mathrm{S}}$. $\quad e_1 = x \qquad T^S_1 = \Gamma(x)$

  If $x \notin \overline{x}$, then $[\overline{e_2}/\overline{x}]e_1 = e_1$ and hence $T^S_3 = T^S_1$.

  Otherwise, $x = x_i$, so $[\overline{e_2}/\overline{x}]e_1 = [\overline{e_2}/\overline{x}]x_i = e_{2_i}$ and $\overline{CD^S}, \Gamma \vdash e_{2_i} :^S T^S_{2_i} = T^S_3$. By the hypothesis $\overline{T^S_2} \leq \overline{T^S_x}$ we have $T^S_2 = T^S_3 \leq T^S_x = T^S_1$.

- Case T-This$^{\mathrm{S}}$. `this` $\notin \overline{x}$, so $[\overline{e_2}/\overline{x}]e_1 = e_1 =$ `this` and $T^S_3 = T^S_1$.

- Case T-FieldC$^{\mathrm{S}}$. $\quad e_1 = e.f \qquad T^S_1 = T^S \qquad \overline{CD^S}, \Gamma \vdash e :^S C$

  By the induction hypothesis, $\overline{CD^S}, \Gamma \vdash [\overline{e_2}/\overline{x}]e :^S T^S_0$ for some $\overline{CD^S} \vdash T^S_0 \leq C$. Then, since T^S_0 is subtype of C, by T-FieldC$^{\mathrm{S}}$ we have $\overline{CD^S}, \Gamma \vdash [\overline{e_2}/\overline{x}]e.f :^S T^S = T^S_1 = T^S_3$.

- Case T-FieldU$^{\mathrm{S}}$. $\quad e_1 = e.f \qquad T^S_1 = T^S_{f_1} \vee \ldots \vee T^S_{f_{n_f}} \qquad \overline{CD^S}, \Gamma \vdash e :^S C_1 \vee \ldots \vee C_n$

  By the induction hypothesis, $\overline{CD^S}, \Gamma \vdash [\overline{e_2}/\overline{x}]e :^S T^S_0$ for some $\overline{CD^S} \vdash T^S_0 \leq C_1 \vee \ldots \vee C_n$. Then, by T-FieldU$^{\mathrm{S}}$, T^S_0 must provide at least one f field such that $\overline{CD^S}, \Gamma \vdash [\overline{e_2}/\overline{x}]e.f :^S T^S_3 \leq T^S_{f_1} \vee \ldots \vee T^S_{f_{n_f}} = T^S_1$.

- Case T-InvC$^{\mathrm{S}}$. $\quad e_1 = e_{obj}.m(\overline{e_{arg}}) \qquad T^S_1 = T^S_r \qquad \overline{CD^S}, \Gamma \vdash e_{obj} :^S C_{obj}$

  By the induction hypothesis, we have $\overline{CD^S}, \Gamma \vdash [\overline{e_2}/\overline{x}]e_{obj} :^S T^S_0$ for some $\overline{CD^S} \vdash T^S_0 \leq C_{obj}$. Likewise, we apply the induction hypothesis to the arguments, obtaining a subtype of the argument type. Then, W-Method$^{\mathrm{S}}$ and $\leq$ ensure that the derived type (T^S_0) provides one appropriate m method with the same exact return and parameter types (W-Method$^{\mathrm{S}}$ states that method overriding is invariant), so $\overline{CD^S}, \Gamma \vdash [\overline{e_2}/\overline{x}](e_{obj}.m(\overline{e_{arg}})) :^S T^S_r = T^S_1$.

– Case T-InvU$^{\text{S}}$. $\quad e_1 = e_{obj}.m(\overline{e_{arg}}) \qquad T_1^S = T_{r_1}^S \vee \ldots \vee T_{r_{nm}}^S$

$\qquad\qquad\qquad\qquad \overline{CD^S}, \Gamma \vdash e_{obj} :^S C_1 \vee \ldots \vee C_n$

By the induction hypothesis, we have $\overline{CD^S}, \Gamma \vdash [\overline{e_2/x}]e_{obj} :^S T_0^S$ for some $\overline{CD^S} \vdash T_0^S \leq C_1 \vee \ldots \vee C_n$. The same applies to the arguments. Then, by W-Method$^{\text{S}}$ we know that T_0^S provides at least one appropriate m method so that $\overline{CD^S}, \Gamma \vdash [\overline{e_2/x}](e_{obj}.m(\overline{e_{arg}})) :^S T_3^S \leq T_{r_1}^S \vee \ldots \vee T_{r_{nm}}^S = T_1^S$.

– Case T-New$^{\text{S}}$. $\quad e_1 = \texttt{new}\, C(\overline{e_{arg}}) \qquad T_1^S = C$

By the induction hypothesis, we have $\overline{CD^S}, \Gamma \vdash [\overline{e_2/x}]e_{obj} :^S T_0^S$ for some $\overline{CD^S} \vdash \overline{T_0^S} \leq \overline{T_{arg}^S}$. Then, since T-Const$^{\text{S}}$ allows the arguments to be subtypes of the parameters, we have $\overline{CD^S}, \Gamma \vdash [\overline{e_2/x}](\texttt{new}\, C(\overline{e_{arg}})) :^S C = T_1^S$.

□

Lemma 7. *$\texttt{this}$ substitution preserves typing in FC#^{S}.*

If $\vdash P^S = \overline{CD^S}\, e_1 :^S \diamond$ and $\overline{CD^S}, \Gamma, \texttt{this}{:}T_{this}^S \vdash e_1 :^S T_1^S$ and $\overline{CD^S}, \Gamma \vdash e_2 :^S T_2^S$ and $\overline{CD^S} \vdash T_2^S \leq T_{this}^S$, then $\overline{CD^S}, \Gamma \vdash [e_2/\texttt{this}]e_1 :^S T_3^S$ for some $\overline{CD^S} \vdash T_3^S \leq T_1^S$.

Proof. By induction on the typing derivation. Similar to Lemma 6. □

Lemma 8. *Preservation of FC#^{S} without warnings.*

If $\vdash P^S = \overline{CD^S} e_1 :^S \diamond$ and $\overline{CD^S}, \varnothing \vdash e_1 :^S T_1^S$ without warnings then, $\overline{CD^S} \vdash e_1 \longrightarrow^S e_2$ and $\overline{CD^S}, \varnothing \vdash e_2 :^S T_2^S$ without warnings for some $\overline{CD^S} \vdash T_2^S \leq T_1^S$.

Proof. By induction on the typing derivation and then selecting the different evaluation rules that can be applied.

– Cases T-Var$^{\text{S}}$ and T-This$^{\text{S}}$. These cases cannot occur because Γ is empty.

– Case T-FieldC$^{\text{S}}$. $\quad e_1 = e.f \qquad T_1^S = T^S \qquad \overline{CD^S}, \Gamma \vdash e :^S C$

  ◦ R-FieldE$^{\text{S}}$. e_1 cannot be reduced to an error, because e_1 would be a value and we know it provides an f field (hypothesis), so R-FieldE$^{\text{S}}$ would not be evaluated.

  ◦ R-FieldC$^{\text{S}}$. For R-FieldC$^{\text{S}}$, the induction hypothesis makes $\overline{CD^S} \vdash e \longrightarrow^{SE} e_3$ and $\overline{CD^S}, \varnothing \vdash e_3 :^S T_3^S \leq C$. Since $T_3^S \leq C$, T_3^S must provide the same f field as C, by definition of the *fields* function. Then $\overline{CD^S}, \varnothing \vdash e_2.f :^S T_2^S \leq T_1^S = T^S$.

  ◦ R-Field$^{\text{S}}$. If the evaluation rule applied is R-Field$^{\text{S}}$, then $e_1 = \texttt{new}\, C(\overline{\sigma_{arg}}).f$. By the definition of T-New$^{\text{S}}$, arguments must $\leq$ the corresponding constructor parameters; by W-Const$^{\text{S}}$, constructor parameters must $\leq$ the corresponding fields. Since T_1^S is the type of the f field, then $\overline{CD^S} \vdash T_2^S \leq T_1^S$.

– Case T-FieldU$^{\text{S}}$. $\quad e_1 = e.f \qquad T_1^S = T_{f_1}^S \vee \ldots \vee T_{f_{n_f}}^S \qquad \overline{CD^S}, \Gamma \vdash e :^S C_1 \vee \ldots \vee C_n$

  ◦ R-FieldC$^{\text{S}}$ and R-FieldE$^{\text{S}}$. By Lemma 3, we know that e is not a value, so neither R-FieldC$^{\text{S}}$ nor R-FieldE$^{\text{S}}$ are reduced.

- R-FIELDC$^{\text{S}}$. By the induction hypothesis, $\overline{CD^S} \vdash e \longrightarrow^S e_3$ and $\overline{CD^S}, \varnothing \vdash e_3 :^S T_3^S \leq C_1 \vee \ldots \vee C_n$. However, this subtyping relation does not ensure that T_3^S provides a suitable f field with the original type system, since T_3^S may be a C_i not providing f. In that case, $e_3.f$ will not be well typed. Nevertheless, the premise of this lemma is that FC#$^{\text{S}}$ produces no warnings. So Lemmas 3 and 4 ensure that all the C_i classes provide an f field, and hence $T_2^S \leq T_1^S = T_{f_1}^S \vee \ldots \vee T_{f_{n_f}}^S$.

– Case T-INVC$^{\text{S}}$. $\quad e_1 = e_{obj}.m(\overline{e_{arg}}) \qquad T_1^S = T_r^S \qquad \overline{CD^S}, \varnothing \vdash e_{obj} :^S C_{obj}$

- R-INVE$^{\text{S}}$, R-PARE$^{\text{S}}$ and R-ARGE$^{\text{S}}$. e_1 is not reduced to an error. In that case, e_{obj} and $\overline{e_{arg}}$ would be values. If so, R-INVE$^{\text{S}}$, R-PARE$^{\text{S}}$ and R-ARGE$^{\text{S}}$ would require that, respectively, m is not provided, a wrong number of parameters is passed, or the argument types are not valid. Since all these predicates are checked as premises of Case T-INVC$^{\text{S}}$, e_1 is not reduced to an error.
- R-INV$^{\text{S}}$. In this case, e_{obj} and $\overline{e_{arg}}$ are values, so $e_1 = \texttt{new}\, C_{obj}(\overline{\sigma_{obj}})$ and $\overline{e_{arg}} = \texttt{new}\, C_{arg_1}(\overline{\sigma_{arg_1}}), \ldots, \texttt{new}\, C_{arg_n}(\overline{\sigma_{arg_n}})$. We know that the type of e_1 (C_{obj}) provides an appropriate m method with e_{body} body, such that $\overline{CD^S}, \Gamma, \overline{x}{:}\overline{T_{arg}^S}, \texttt{this}{:}C_{obj} \vdash e_{body} :^S T_{body}^S$. By Lemmas 6 and 7, we know that $\overline{CD^S}, \varnothing \vdash [\texttt{new}\, C_{arg_1}(\overline{\sigma_{arg_1}})/x_1\ , \ \ldots\ ,\ \texttt{new}\, C_{arg_n}(\overline{\sigma_{arg_n}})/x_n\ ,\ \texttt{new}\, C_{obj}(\overline{\sigma_{obj}})/\texttt{this}]e_{body} :^S T_2^S$ for some $\overline{CD^S} \vdash T_2^S \leq T_{body}^S$. Then, by W-METHOD$^{\text{S}}$, we know that $T_{body}^S \leq T_r^S = T_1^S$.
- R-INVC$^{\text{S}}$. R-INVC$^{\text{S}}$. If e is not a value, then R-INVC$^{\text{S}}$ will be reduced. By the induction hypothesis, we know that $\overline{CD^S} \vdash e_{obj} \longrightarrow^S e_3$ and $\overline{CD^S}, \varnothing \vdash e_3 :^S T_3^S \leq C_{obj}$. By W-METHOD$^{\text{S}}$, we know T_3^S provides an m method with the same signature as m in C_{obj}. Then, since the arguments are subtypes of the parameters, $T_2^S = T_r^S = T_1^S$.
- R-INVAC$^{\text{S}}$. If e_{obj} is a value, but one the expressions in $\overline{e_{arg}}$ is not, R-INVAC$^{\text{S}}$ will be reduced. Applying the induction hypothesis, as in the previous subcase, we have $T_2^S = T_r^S = T_1^S$.

– Case T-INVU$^{\text{S}}$. $\quad e_1 = e_{obj}.m(\overline{e_{arg}}) \qquad T_1^S = T_{r_1}^S \vee \ldots \vee T_{r_{n_m}}^S$
$\overline{CD^S}, \varnothing \vdash e_{obj} :^S C_1 \vee \ldots \vee C_n$

- R-INV$^{\text{S}}$, R-INVE$^{\text{S}}$, R-PARE$^{\text{S}}$ and R-ARGE$^{\text{S}}$. By Lemma 3, we know that e_{obj} is not a value, so these rules cannot be applied.
- R-INVC$^{\text{S}}$. If e is not a value, then R-INVC$^{\text{S}}$ will be reduced. By the induction hypothesis, we know that $\overline{CD^S} \vdash e_{obj} \longrightarrow^S e_3$ and $\overline{CD^S}, \varnothing \vdash e_3 :^S T_3^S \leq C_1 \vee \ldots \vee C_n$. By W-METHOD$^{\text{S}}$, we know that T_3^S provides an m method with the same signature as at least one m in C_i. As for field access, this condition does not ensure that T_3^S provides the expected m method if warnings are not considered as type errors. That is the reason why the hypothesis of the lemma requires warnings as errors. Then, by Lemmas 3 and 5, we can prove that $T_2^S = T_r^S = T_1^S$.
- R-INVAC$^{\text{S}}$. Proved the same as in the previous subcase.

– Case T-New$^{\mathrm{S}}$. $\quad e_1 = \texttt{new}\, C(\overline{e_{arg_1}}, \overline{e_{arg_2}}) \qquad T^S_1 = C$

If $\overline{e_{arg_1}}$ and $\overline{e_{arg_2}}$ are values, so is e_1. Otherwise, R-NewAC$^{\mathrm{S}}$ is evaluated. Let e_{arg_i} be the reduced term. By the induction hypothesis, $\overline{CD^S} \vdash e_{arg_i} \longrightarrow^S e_3$ and $\overline{CD_S}, \varnothing \vdash e_3 :^S T^S_3 \leq T^S_{arg_i}$. By the definition of W-Const$^{\mathrm{S}}$, we have that $T^S_2 = C = T^S_1$.

□

Property 4. *Static type safety of FC#^{S} when warnings are considered as errors.*

If $\vdash P^S = \overline{CD^S}\, e_1 :^S \diamond$ and $\overline{CD^S}, \varnothing \vdash e_1 :^S T^S_1$ without any warning, then

- either $e_1 \in$ Values,
- or $\exists\, e_2 \,.\, \overline{CD_G} \vdash e_1 \longrightarrow^S e_2$ and $\overline{CD_S}, \varnothing \vdash e_2 :^S T^S_2$ for some $\overline{CD_S} \vdash T^S_2 \leq T^S_1$ without any warning

Proof. By induction on the evaluation steps.

For the base case, where $e_1 = e_2$, Progress (Property 3) tells us that e_1 is either a value, it can be evaluated, or is reduced to an error. The last option is not applicable since it is well typed and Preservation (Lemma 8) prevents this from occurring. For the case that $\overline{CD^S} \vdash e_1 \longrightarrow^{S*} e_2$ and $\overline{CD^G} \vdash e_2 \longrightarrow^S e_3$, e_2 is well-typed by the induction hypothesis, and e_3 will be well-typed by Progress. □

Appendix J. Specialize functions

$$\frac{\begin{array}{c} P^G = \overline{CD^G_p}\, e_p \qquad T^G_{pr}\, m(\overline{T^G_{pp}\, x})\texttt{\{}\,\texttt{return}\, e_{pbody}\,\texttt{;\}} \in \mathit{methods}(\overline{CD^G_p}, C_{obj}, m) \\ \forall T^S_{pp_i} \in \overline{T^S_{pp}} \,.\, T^S_{param_i} = T^G_{pp_i} \text{ is dynamic ? } T^S_{arg_i} : T^G_{pp_i} \\ \overline{T^S_{param}} \to T^S_r \notin \mathit{types}(\mathit{methods}(\overline{CD^S_{in}}, C_{obj}, m)) \qquad \overline{C_{sub}} = C_{obj}, \mathit{subclasses}(\overline{CD^S_{in}}, C_{obj}) \\ n = \mathit{newmethod}(\overline{CD^S_{in}}, \overline{C_{sub}}, m) \qquad \overline{CD^{S\prime}_0} = \overline{CD^S_{in}} \qquad n_{sub} = \#(\overline{C^S_{sub}}) \\ \forall i \in [1, n_{sub}] \,.\, \overline{CD^{S\prime}_{spe_i}} = \mathit{specializemn}(P^G, \overline{CD^{S\prime}_{i-1}}, \Gamma, C_{sub_i}, m, \overline{T^S_{arg}}, n) \qquad \overline{CD^S_{out}} = \overline{CD^{S\prime}_{n_{sub}}} \end{array}}{\mathit{specializem}(P^G, \overline{CD^S_{in}}, \Gamma, C_{obj}, m, \overline{T^S_{arg}}) = \overline{CD^S_{out}}, n}$$

$$\frac{P^G = \overline{CD^G_p}\, e_p \qquad T^G_{pr}\, m(\overline{T^G_{pp}\, x})\texttt{\{}\,\texttt{return}\, e_{pbody}\,\texttt{;\}} \notin \mathit{methods}(\overline{CD^G_p}, C_{obj}, m)}{\mathit{specializem}(P^G, \overline{CD^S_{in}}, \Gamma, C_{obj}, m, \overline{T^S_{arg}}) = \overline{CD^S_{in}}, 0}$$

$$\frac{\begin{array}{c} P^G = \overline{CD^G_p}\, e_p \qquad T^G_{pr}\, m(\overline{T^G_{pp}\, x})\texttt{\{}\,\texttt{return}\, e_{pbody}\,\texttt{;\}} \in \mathit{methods}(\overline{CD^G_p}, C_{obj}, m) \\ \forall T^S_{pp_i} \in \overline{T^S_{pp}} \,.\, T^S_{param_i} = T^G_{pp_i} \text{ is dynamic ? } T^S_{arg_i} : T^G_{pp_i} \\ T^S_r\; m\_n(\overline{T^S_{param}\, x})\texttt{\{}\,\texttt{return}\, e_{body}\,\texttt{;\}} \in \mathit{methods}(\overline{CD^S_{in}}, C_{obj}, m) \end{array}}{\mathit{specializem}(P^G, \overline{CD^S_{in}}, \Gamma, C_{obj}, m, \overline{T^S_{arg}}) = \overline{CD^S_{in}}, n}$$

$$\frac{\begin{array}{c} P^G = \overline{CD^G_p}\, e_p \qquad T^G_{pr}\, m(\overline{T^G_{pp}\, x})\texttt{\{}\,\texttt{return}\, e_{pbody}\,\texttt{;}\texttt{\}} \in \mathit{methods}(\overline{CD^G_p}, C_{obj}, m) \\ \forall\, T^S_{pp_i} \in \overline{T^S_{pp}}\,.\, T^S_{param_i} = T^G_{pp_i} \text{ is dynamic ? } T^S_{arg_i} : T^G_{pp_i} \\ \overline{T^S_{param}} \to T^S_r \notin \mathit{types}(\mathit{methods}(\overline{CD^S_{in}}, C_{obj}, m)) \\ \Gamma_m = \overline{x{:}T^S_{param}}, \texttt{this}{:}C_{obj}, \Gamma \qquad P^G, \overline{CD^S_{in}}, \Gamma_m \vdash e_{pbody} \Rightarrow \langle e'_{body}, \overline{CD^S_{spe}} \rangle \\ \overline{CD^S_{spe}}, \Gamma_m \vdash e'_{body} :^S T^S_{body} \qquad T^S_r = T^G_{pr} \text{ is dynamic ? } T^S_{body} : T^G_{pr} \\ M^S = T^S_r\, m\_n(\overline{T^S_{param}\, x})\texttt{\{}\,\texttt{return}\, e'_{body}\,\texttt{;}\texttt{\}} \qquad \overline{CD^S_{out}} = \mathit{addmethod}(\overline{CD^S_{spe}}, C_{obj}, M^S) \end{array}}{\mathit{specializemn}(P^G, \overline{CD^S_{in}}, \Gamma, C_{obj}, m, \overline{T^S_{arg}}, n) = \overline{CD^S_{out}}}$$

$$\frac{\begin{array}{c} P^G = \overline{CD^G_p}\, e_p \qquad T^G_{pr}\, m(\overline{T^G_{pp}\, x})\texttt{\{}\,\texttt{return}\, e_{pbody}\,\texttt{;}\texttt{\}} \in \mathit{methods}(\overline{CD^G_p}, C_{obj}, m) \\ \forall\, T^S_{pp_i} \in \overline{T^S_{pp}}\,.\, T^S_{param_i} = T^G_{pp_i} \text{ is dynamic ? } T^S_{arg_i} : T^G_{pp_i} \\ \overline{T^S_{param}} \to T^S_r \notin \mathit{types}(\mathit{methods}(\overline{CD^S_{in}}, C_{obj}, m)) \\ \Gamma_m = \overline{x{:}T^S_{param}}, \texttt{this}{:}C_{obj}, \Gamma \qquad P^G, \overline{CD^S_{in}}, \Gamma_m \vdash e_{pbody} \Rightarrow \langle e'_{body}, \overline{CD^S_{spe}} \rangle \\ \mathit{not}(\overline{CD^S_{spe}}, \Gamma_m \vdash e'_{body} :^S T^S_{body}) \qquad T^S_r = T^G_{pr} \text{ is dynamic ? Object } : T^G_{pr} \\ M^S = T^S_r\, m\_n(\overline{T^S_{param}\, x})\texttt{\{}\,\texttt{return}\, e'_{body}\,\texttt{;}\texttt{\}} \qquad \overline{CD^S_{out}} = \mathit{addmethod}(\overline{CD^S_{spe}}, C_{obj}, M^S) \end{array}}{\mathit{specializemn}(P^G, \overline{CD^S_{in}}, \Gamma, C_{obj}, m, \overline{T^S_{arg}}, n) = \overline{CD^S_{out}}}$$

$$\frac{\begin{array}{c} P^G = \overline{CD^G_p}\, e_p \qquad T^G_{pr}\, m(\overline{T^G_{pp}\, x})\texttt{\{}\,\texttt{return}\, e_{pbody}\,\texttt{;}\texttt{\}} \in \mathit{methods}(\overline{CD^G_p}, C_{obj}, m) \\ \forall\, T^S_{pp_i} \in \overline{T^S_{pp}}\,.\, T^S_{param_i} = T^G_{pp_i} \text{ is dynamic ? } T^S_{arg_i} : T^G_{pp_i} \\ T^S_r\, m\_k(\overline{T^S_{param}\, x})\texttt{\{}\,\texttt{return}\, e_{body}\,\texttt{;}\texttt{\}} \in \mathit{methods}(\overline{CD^S_{in}}, C_{obj}, m) \\ M^S = T^S_r\, m\_n(\overline{T^S_{param}\, x})\texttt{\{}\,\texttt{return}\, e_{body}\,\texttt{;}\texttt{\}} \qquad \overline{CD^S_{out}} = \mathit{addmethod}(\overline{CD^S_{in}}, C_{obj}, M^S) \end{array}}{\mathit{specializemn}(P^G, \overline{CD^S_{in}}, \Gamma, C_{obj}, m, \overline{T^S_{arg}}, n) = \overline{CD^S_{out}}}$$

$$\frac{\begin{array}{c} P^G = CD^G_1 \ldots \texttt{class}\, C : C_{super}\, \texttt{\{}\, \overline{T^G_f\, f}\,\texttt{;}\; K^G_p\; \overline{M^G_p}\, \texttt{\}} \ldots CD^G_n\; e_p \\ K^G_p = C(\overline{T^G_{psuper}\, f_{super}}, \overline{T^G_{psub}\, f_{sub}})\texttt{:}\,\texttt{base(}\overline{f_{super}}\texttt{)}\,\texttt{\{}\,\overline{\texttt{this.}f_{sub} = f_{sub}\texttt{;}}\texttt{\}} \\ \overline{CD^S_{super}} = \mathit{specializec}(P^G, \overline{CD^S_{in}}, \Gamma, C_{super}, \overline{e_{arg_{super}}}) \\ \overline{CD^S_{kf}}, K^S, \overline{T^S_{field}} = \mathit{specializekf}(P^S, \overline{CD^S_{super}}, \Gamma, C, \overline{e_{arg_{super}}}, \overline{e_{arg_{sub}}}) \\ \overline{CD^S_{out}} = \overline{CD^S_{kf}}, \texttt{class}\, C : C_{super}\, \texttt{\{}\, \overline{T^S_{field} f}\,\texttt{;}\; K^S\; \overline{M^S}\, \texttt{\}} \end{array}}{\mathit{specializec}(P^G, \overline{CD^S_{in}}, \Gamma, C, \overline{e_{arg_{super}}}, \overline{e_{arg_{sub}}}) = \overline{CD^S_{out}}}$$

$$\frac{}{\mathit{specializec}(P^G, \overline{CD^S_{in}}, \varnothing, \mathit{Object}, \varnothing, \varnothing) = \overline{CD^S_{in}}}$$

$$\frac{\begin{array}{c}
C \notin \overline{CD^S_{in}} \qquad P^G = CD^G_1 \ldots \texttt{class}\, C : C_{super}\, \texttt{\{}\, \overline{T^G_f f}\,\texttt{;}\, K^G_p\, \overline{M^G_p}\, \texttt{\}} \ldots CD^G_m\, e_p \\
K^G_p = C(\overline{T^G_{psuper}\, f_{super}}, \overline{T^G_{psub}\, f_{sub}})\texttt{: base(}\overline{f_{super}}\texttt{) \{}\overline{\texttt{this.}f_{sub}} = \overline{f_{sub}}\texttt{;\}} \\
\forall\, e_{arg_{super_i}} \in \overline{e_{arg_{super}}} \,.\, \overline{CD^S_{in}}, \Gamma \vdash e_{arg_{super_i}} :^S T^S_{arg_{super_i}} \\
\forall\, e_{arg_{sub_i}} \in \overline{e_{arg_{sub}}} \,.\, \overline{CD^S_{in}}, \Gamma \vdash e_{arg_{sub_i}} :^S T^S_{arg_{sub_i}} \\
\forall\, T^G_{psuper_i} \in \overline{T^G_{psuper}} \,.\, T^G_{parsuper_i} = T^G_{psuper_i} \text{ is dynamic ? } T^S_{argsuper_i} : T^G_{psuper_i} \\
\forall\, T^G_{psub_i} \in \overline{T^G_{psub}} \,.\, T^G_{parsub_i} = T^G_{psub_i} \text{ is dynamic ? } T^S_{argsub_i} : T^G_{psub_i} \\
K^S = C(\overline{T^S_{parsuper}\, f_{super}}, \overline{T^S_{parsub}\, f_{sub}})\texttt{: base(}\overline{f_{super}}\texttt{) \{}\overline{\texttt{this.}f_{sub}} = \overline{f_{sub}}\texttt{;\}} \\
\forall\, T^G_{psub_i} \in \overline{T^G_{psub}} \,.\, T^S_{field_i} = T^G_{f_i} \text{ is dynamic ? } T^S_{argsub_i} : T^G_{f_i}
\end{array}}{\mathit{specializekf}(P^G, \overline{CD^S_{in}}, \Gamma, C, \overline{e_{arg_{super}}}, \overline{e_{arg_{sub}}}) = \overline{CD^S_{in}}, K^S, \overline{T^S_{field}}}$$

$$\frac{\begin{array}{c}
\overline{CD^S_{in}} = CD^S_1 \ldots \texttt{class}\, C : C_{super}\, \texttt{\{}\, \overline{T^S_f f}\,\texttt{;}\, K^S_p\, \overline{M^S_p}\, \texttt{\}} \ldots CD^S_m \\
K^S_p = C(\overline{T^S_{psuper}\, f_{super}}, \overline{T^S_{psub}\, f_{sub}})\texttt{: base(}\overline{f_{super}}\texttt{) \{}\overline{\texttt{this.}f_{sub}} = \overline{f_{sub}}\texttt{;\}} \\
P^G = CD^G_1 \ldots \texttt{class}\, C : C_{super}\, \texttt{\{}\, \overline{T^G_f f}\,\texttt{;}\, K^G_p\, \overline{M^G_p}\, \texttt{\}} \ldots CD^G_m\, e_p \\
K^G_p = C(\overline{T^G_{psuper}\, f_{super}}, \overline{T^G_{psub}\, f_{sub}})\texttt{: base(}\overline{f_{super}}\texttt{) \{}\overline{\texttt{this.}f_{sub}} = \overline{f_{sub}}\texttt{;\}} \\
\forall\, e_{arg_{super_i}} \in \overline{e_{arg_{super}}} \,.\, \overline{CD^S_{in}}, \Gamma \vdash e_{arg_{super_i}} :^S T^S_{arg_{super_i}} \\
\forall\, e_{arg_{sub_i}} \in \overline{e_{arg_{sub}}} \,.\, \overline{CD^S_{in}}, \Gamma \vdash e_{arg_{sub_i}} :^S T^S_{arg_{sub_i}} \\
\forall\, T^G_{psuper_i} \in \overline{T^G_{psuper}} \,.\, T^G_{parsuper_i} = T^G_{psuper_i} \text{ is dynamic ? } T^S_{psuper_i} \vee T^S_{argsuper_i} : T^G_{psuper_i} \\
\forall\, T^G_{psub_i} \in \overline{T^G_{psub}} \,.\, T^G_{parsub_i} = T^G_{psub_i} \text{ is dynamic ? } T^S_{psub_i} \vee T^S_{argsub_i} : T^G_{psub_i} \\
K^S = C(\overline{T^S_{parsuper}\, f_{super}}, \overline{T^S_{parsub}\, f_{sub}})\texttt{: base(}\overline{f_{super}}\texttt{) \{}\overline{\texttt{this.}f_{sub}} = \overline{f_{sub}}\texttt{;\}} \\
\forall\, T^G_{psub_i} \in \overline{T^G_{psub}} \,.\, T^S_{field_i} = T^G_{f_i} \text{ is dynamic ? } T^S_{f_i} \vee T^S_{argsub_i} : T^G_{f_i}
\end{array}}{\mathit{specializekf}(P^G, \overline{CD^S_{in}}, \Gamma, C, \overline{e_{arg_{super}}}, \overline{e_{arg_{sub}}}) = \overline{CD^S_{in}}, K^S, \overline{T^S_{field}}}$$

$$\frac{\begin{array}{c}
\overline{e_{arg}} = \overline{e_{arg_{super}}}, \overline{e_{arg_{sub}}} \qquad \exists\, e_{arg_i} \in \overline{e_{arg}} \,.\, \mathit{not}(\overline{CD^S_{in}}, \Gamma \vdash e_{arg_i} :^S T^S_{arg_i}) \\
C \notin \overline{CD^S_{in}} \qquad P^G = CD^G_1 \ldots \texttt{class}\, C : C_{super}\, \texttt{\{}\, \overline{T^G_f f}\,\texttt{;}\, K^G_p\, \overline{M^G_p}\, \texttt{\}} \ldots CD^G_m\, e_p \\
K^G_p = C(\overline{T^G_{psuper}\, f_{super}}, \overline{T^G_{psub}\, f_{sub}})\texttt{: base(}\overline{f_{super}}\texttt{) \{}\overline{\texttt{this.}f_{sub}} = \overline{f_{sub}}\texttt{;\}} \\
\forall\, T^G_{psuper_i} \in \overline{T^G_{psuper}} \,.\, T^G_{parsuper_i} = T^G_{psuper_i} \text{ is dynamic ? Object : } T^G_{psuper_i} \\
\forall\, T^G_{psub_i} \in \overline{T^G_{psub}} \,.\, T^G_{parsub_i} = T^G_{psub_i} \text{ is dynamic ? Object : } T^G_{psub_i} \\
K^S = C(\overline{T^S_{parsuper}\, f_{super}}, \overline{T^S_{parsub}\, f_{sub}})\texttt{: base(}\overline{f_{super}}\texttt{) \{}\overline{\texttt{this.}f_{sub}} = \overline{f_{sub}}\texttt{;\}} \\
\forall\, T^G_{psub_i} \in \overline{T^G_{psub}} \,.\, T^S_{field_i} = T^G_{f_i} \text{ is dynamic ? Object : } T^G_{f_i}
\end{array}}{\mathit{specializekf}(P^G, \overline{CD^S_{in}}, \Gamma, C, \overline{e_{arg_{super}}}, \overline{e_{arg_{sub}}}) = \overline{CD^S_{in}}, K^S, \overline{T^S_{field}}}$$

$$\frac{\begin{array}{c}
\exists\, e_{arg_i} \in \overline{e_{arg}} \,.\, \mathit{not}(\overline{CD^S_{in}}, \Gamma \vdash e_{arg_i} :^S T^S_{arg_i}) \\
\overline{CD^S_{in}} = CD^S_1 \ldots \texttt{class}\, C : C_{super}\, \texttt{\{}\, \overline{T^S_f f}\,\texttt{;}\, K^S_p\, \overline{M^S_p}\, \texttt{\}} \ldots CD^S_m
\end{array}}{\mathit{specializekf}(P^G, \overline{CD^S_{in}}, \Gamma, C, \overline{e_{arg}}) = \overline{CD^S_{in}}, K^S_p, \overline{T^S_f}}$$

Appendix K. Additional functions used in the specialization

$$\frac{\begin{array}{c}\overline{CD_{in}} = CD_1 \ldots \texttt{class}\, C : C_{super}\, \texttt{\{}\, \overline{T_f f}\, \texttt{;}\, K\, \overline{M_c}\, \texttt{\}} \ldots CD_m \\ \overline{M_{sub}} = \{M \,.\, M = T_r\, m\_n(\overline{T_p^G\, x})\texttt{\{}\,\texttt{return}\, e\texttt{;}\texttt{\}} \in \overline{M_c}\} \\ \overline{M_{super}} = \{M_{super} \,.\, M_{super} \notin \overline{M_{sub}} \text{ and } M_{super} \in methods(\overline{CD_{in}}, C_{super}, m)\}\end{array}}{methods(\overline{CD_{in}}, C, m) = \overline{M_{sub}} \cup \overline{M_{super}}}$$

$$\frac{}{methods(\overline{CD_{in}}, Object, m) = \varnothing} \qquad \frac{}{types(\overline{M}) = \{\overline{T_p} \to T_r \,.\, T_r\, m(\overline{T_p\, x})\texttt{\{}\,\texttt{return}\, e\texttt{;}\texttt{\}} \in \overline{M}}$$

$$\frac{\begin{array}{c}\overline{CD^S_{in}} = CD^S_1 \ldots \texttt{class}\, C : C_{super}\, \texttt{\{}\, \overline{T^S_f f}\, \texttt{;}\, K^S_p\, \overline{M^S_c}\, \texttt{\}} \ldots CD^S_n \\ M^S \notin \overline{M^S_c} \qquad \overline{CD^S_{out}} = CD^S_1 \ldots \texttt{class}\, C : C_{super}\, \texttt{\{}\, \overline{T^S_f f}\, \texttt{;}\, K^S_p\, \overline{M^S_c}, M^S\, \texttt{\}} \ldots CD^S_n\end{array}}{addmethod(\overline{CD^S_{in}}, C, M^S) = \overline{CD^S_{out}}}$$

$$\frac{\overline{CD^S_{in}} = CD^S_1 \ldots \texttt{class}\, C : C_{super}\, \texttt{\{}\, \overline{T^S_f f}\, \texttt{;}\, K^S_p\, \overline{M^S_c}\, \texttt{\}} \ldots CD^S_n \qquad M^S \in \overline{M^S_c}}{addmethod(\overline{CD^S_{in}}, C, M^S) = \overline{CD^S_{in}}}$$

$$\frac{\begin{array}{c}\overline{CD^S_{in}} = CD^S_1 \ldots \texttt{class}\, C : C_{super}\, \texttt{\{}\, \overline{T^S_f f}\, \texttt{;}\, K^S\, \overline{M^S_c}\, \texttt{\}} \ldots CD^S_n \\ \overline{M^S} = M^S_1 \ldots T^S_{r_1}\, m\_1(\overline{T^S_{p_1}\, x})\texttt{\{}\,\texttt{return}\, e_1\texttt{;}\texttt{\}} \ldots T^S_{r_n}\, m\_n(\overline{T^S_{p_n}\, x})\texttt{\{}\,\texttt{return}\, e_n\texttt{;}\texttt{\}} \ldots M^S_q\end{array}}{newmethod(\overline{CD^S_{in}}, C, m) = n + 1}$$

$$\frac{\forall\, C_i \in \overline{C} \,.\, n_i = newmethod(\overline{CD^S_{in}}, C_i, m) \qquad n = max(n_i)}{newmethod(\overline{CD^S_{in}}, \overline{C}, m) = n}$$

$$\frac{}{subclasses(\overline{CD^S_{in}}, C) = \{C_{sub} \,.\, \texttt{class}\, C_{sub} : C\, \texttt{\{}\, \overline{T_f f}\, \texttt{;}\, K^S\, \overline{M^S}\, \texttt{\}} \in \overline{CD^S_{in}}\}}$$

Appendix L. Well-typed FC#$^{\mathbf{S}}$ programs can always be specialized

Property 5. *Well-typed FC#^{G} programs can always be specialized (and termination of the specialization is ensured).*

$\forall\ \vdash P^G = \overline{CD^G}\, e_1 :^G \diamond$ and $\overline{CD^G}, \varnothing \vdash e_1 :^G T^G_1 \,.\, \exists\, e_2, \overline{CD^S}$ such that $P^G, \varnothing, \varnothing \vdash e_1 \Rightarrow \langle e_2, \overline{CD^S}\rangle$.

Proof. By induction on the typing derivation of FC#$^{\text{G}}$. We check that the expression can be specialized and that specialization terminates.

– Cases T-Var$^{\text{G}}$ and T-This$^{\text{G}}$. The corresponding SP-Var and SP-This rules have no premises, so they can always be applied. There are no subexpressions, so termination is ensured.

– Cases T-FieldC$^{\text{G}}$ and T-FieldD$^{\text{G}}$. $e_1 = e.f$

The only specialization rule to be applied in these cases is SP-Field. In that rule, the only premise is that e could be specialized, which is indeed the induction hypothesis. Specialization terminates, since it is not possible to have recursive subexpressions.

– Case T-Inv$^{\text{G}}$. $e_1 = e_{obj}.m(\overline{e_{arg}})$

By the induction hypothesis, e_{obj} can be specialized to e'_{obj} (the same applies to the arguments). Then, it could happen that e'_{obj} in FC#$^{\text{S}}$ is well typed to 1) a class, 2) a union type or 3) it is not well typed. For the first two cases, SP-InvC and SP-InvU require the specialization of the arguments to be well typed in FC#$^{\text{S}}$. Therefore, SP-InvU specializes the program when the previous requirements are not fulfilled. The last premise is that *specializem* and *specializemn* functions, for the two first cases, are evaluated.

The only premise of *specializem* is that the specialized method exists in the original program. Since the program is well typed, that condition is ensured by T-InvC. For *specializemn*, the body of the invoked method is specialized. However, the specialization could be 1) well- or 2) ill-typed. Note that there are two implementations of *specializemn*: for case 1), `dynamic` is replaced with the inferred type; for case 2), it is replaced with `Object`.

Regarding termination, there could be recursive invocations. In that case, the *specializem* function provides an implementation when the method has already been specialized. That implementation does not specialize the method body any more, returning the existing specialization. In this way, termination is ensured.

– Case T-InvD$^{\text{G}}$. $e_1 = e_{obj}.m(\overline{e_{arg}})$

Similar to the previous case. The proof is different when demonstrating that *specializem* can always be evaluated. Since the type of e_{obj} is `dynamic`, the program may not have a definition of the invoked method. For that reason, there is one implementation of *specializem* considering that. That implementation does not specialize the method implementation, returning a non-existent 0 version (a compilation error will be shown by the type system of FC#$^{\text{S}}$).

The proof of termination is similar to the previous case.

– Case T-New$^{\text{G}}$. $e_1 = \texttt{new}\,(\overline{e_{arg1}}, \overline{e_{arg2}})$

For this case, the only specialization rule that can be applied is SP-New. One premise is that the arguments could be evaluated, which is the specialization hypothesis. The other premises are those required by the *specializec* function. That function requires that the class and constructor of C and its superclass must be defined in the program. Since e_1 is well typed, T-New$^{\text{G}}$ and W-Const ensure that premise. Then, *specializec* calls *specializekf*. The only additional premise required by this function is that the specialized arguments are well typed in FC#$^{\text{S}}$. Should this not occur, there are two more implementations of *specializekf* considering this special case (whether or not the method has been previously specialized).

$$\frac{e_1 = e_2}{e_1 \equiv e_2} \qquad \frac{e_1 \equiv e_2}{e_1.f \equiv e_2.f} \qquad \frac{e_1 \equiv e_2 \qquad \overline{e_{arg1}} \equiv \overline{e_{arg2}}}{e_1.m(\overline{e_{arg1}}) \equiv e_2.m(\overline{e_{arg2}})} \qquad \frac{e_1 \equiv e_2 \qquad \overline{e_{arg1}} \equiv \overline{e_{arg2}}}{e_1.m(\overline{e_{arg1}}) \equiv e_2.m\_n(\overline{e_{arg2}})}$$

$$\frac{\overline{e_{arg1}} \equiv \overline{e_{arg2}}}{\texttt{new}\, C(\overline{e_{arg1}}) \equiv \texttt{new}\, C(\overline{e_{arg2}})}$$

Figure M.1: Definition of expression equivalence ($\equiv$).

Specialization terminates, since there are no recursive subexpressions.

□

Appendix M. Program specialization preserves semantics

In Figure M.1 we define the equivalence relation as $\equiv$. It gives the idea of those expressions that can be evaluated to equivalent terms. We will use that relation to prove the semantic equivalence of specialized programs and the original ones.

Lemma 9. *The specialization of one expression is equivalent to the original expression.*

If $\vdash P^G = \overline{CD^G}\, e_m :^G \diamond$ and $\overline{CD^G}, \varnothing \vdash e_m :^G T_m^G$ and $P^G, \varnothing, \varnothing \vdash e_1 \Rightarrow \langle e_2, \overline{CD^S} \rangle$, then $e_1 \equiv e_2$.

Proof. By induction on the specialization rules.

– Cases SP-Var and SP-This. $e_1 = e_2$ so $e_1 \equiv e_2$.

– Case SP-Field. $\quad e_1 = e.f \qquad P^G, \varnothing, \varnothing \vdash e \Rightarrow \langle e', \overline{CD^S} \rangle$

By definition of SP-Field, we know that $e_2 = e_3.f$. By the induction hypothesis, we have that $e \equiv e'$, so, by definition of $\equiv$, we have that $e.f = e_1 \equiv e_2 = e'.f$.

– Cases SP-InvC and SP-InvU. $\quad e_1 = e_{obj}.m(\overline{e_{arg}}) \qquad P^G, \varnothing, \varnothing \vdash e_{obj} \Rightarrow \langle e'_{obj}, \overline{CD^S_{obj}} \rangle$

$$P^G, \overline{CD^S_{obj}}, \varnothing \vdash \overline{e_{arg}} \Rightarrow \langle \overline{e'_{arg}}, \overline{CD^S} \rangle$$

By definition of SP-InvC, we know that $e_2 = e'_{obj}.m\_n(\overline{e'_{arg}})$. By the induction hypothesis, we know that $e_{obj} \equiv e'_{obj}$ and $\overline{e_{arg}} \equiv \overline{e'_{arg}}$. Thus, by definition of $\equiv$, we have that $e_{obj}.m(\overline{e_{arg}}) = e_1 \equiv e_2 = e'_{obj}.m\_n(\overline{e'_{arg}})$.

– Case SP-InvE. $\quad e_1 = e_{obj}.m(\overline{e_{arg}}) \qquad P^G, \varnothing, \varnothing \vdash e_{obj} \Rightarrow \langle e'_{obj}, \overline{CD^S_{obj}} \rangle$

$$P^G, \overline{CD^S_{obj}}, \varnothing \vdash \overline{e_{arg}} \Rightarrow \langle \overline{e'_{arg}}, \overline{CD^S} \rangle$$

In this case, $e_2 = e'_{obj}.m(\overline{e'_{arg}})$. By the induction hypothesis, we know that $e_{obj} \equiv e'_{obj}$ and $\overline{e_{arg}} \equiv \overline{e'_{arg}}$. Therefore, by definition of $\equiv$, we have that $e_{obj}.m(\overline{e_{arg}}) = e_1 \equiv e_2 = e'_{obj}.m(\overline{e'_{arg}})$.

– Case SP-NEW. $e_1 = \texttt{new}\, C(\overline{e_{arg}})$ $P^G, \varnothing, \varnothing \vdash \overline{e_{arg}} \Rightarrow \langle \overline{e'_{arg}}, \overline{CD^S} \rangle$

By definition of SP-NEW, we have that $e_2 = \texttt{new}\, C(\overline{e'_{arg}})$. By the induction hypothesis, $\overline{e_{arg}} \equiv \overline{e'_{arg}}$. Then, by definition of $\equiv$, we have that $\texttt{new}\, C(\overline{e_{arg}}) = e_1 \equiv e_2 = \texttt{new}\, C(\overline{e'_{arg}})$.

□

Lemma 10. *Substitution preserves equivalence.*

If $e_1 \equiv e_2$, then $[\overline{\sigma_{arg}}/\overline{x}, \sigma_{obj}/\texttt{this}]e_1 \equiv [\overline{\sigma_{arg}}/\overline{x}, \sigma_{obj}/\texttt{this}]e_2$.

Proof. By induction on the syntax of expressions.

– Cases $e_1 ::= x$ and $e_1 ::= \texttt{this}$. By definition of $\equiv$, e_2 must be equal to e_1, so the conclusion holds.

– Case $e_1 ::= e_3.f$. By definition of $\equiv$, $e_2 = e_4.f$. By the induction hypothesis, we have $[\overline{\sigma_{arg}}/\overline{x}, \sigma_{obj}/\texttt{this}]e_3 \equiv [\overline{\sigma_{arg}}/\overline{x}, \sigma_{obj}/\texttt{this}]e_4$, so $[\overline{\sigma_{arg}}/\overline{x}, \sigma_{obj}/\texttt{this}]e_3.f \equiv [\overline{\sigma_{arg}}/\overline{x}, \sigma_{obj}/\texttt{this}]$ $e_4.f$.

– Case $e_1 ::= e_{obj1}.m(\overline{e_{arg1}})$. By definition of $\equiv$, $e_2 = e_{obj2}.m(\overline{e_{arg2}})$ or $e_2 = e_{obj2}.m\_n(\overline{e_{arg2}})$. By the induction hypothesis, we have $[\overline{\sigma_{arg}}/\overline{x}, \sigma_{obj}/\texttt{this}]e_{obj1} \equiv [\overline{\sigma_{arg}}/\overline{x}, \sigma_{obj}/\texttt{this}]e_{obj2}$ and $[\overline{\sigma_{arg}}/\overline{x}, \sigma_{obj}/\texttt{this}]\overline{e_{arg1}} \equiv [\overline{\sigma_{arg}}/\overline{x}, \sigma_{obj}/\texttt{this}]\overline{e_{arg2}}$. So, by the definition of $\equiv$, we have that $[\overline{\sigma_{arg}}/\overline{x}, \sigma_{obj}/\texttt{this}]e_{obj1}.m(\overline{e_{arg1}}) \equiv [\overline{\sigma_{arg}}/\overline{x}, \sigma_{obj}/\texttt{this}]e_{obj2}.m(\overline{e_{arg2}})$ and $[\overline{\sigma_{arg}}/\overline{x}, \sigma_{obj}/\texttt{this}]$ $e_{obj1}.m(\overline{e_{arg1}}) \equiv [\overline{\sigma_{arg}}/\overline{x}, \sigma_{obj}/\texttt{this}]e_{obj2}.m\_n(\overline{e_{arg2}})$.

– Case $e_1 ::= \texttt{new}\, C(\overline{e_{arg1}})$. By definition of $\equiv$, $e_2 = \texttt{new}\, C(\overline{e_{arg2}})$. By the induction hypothesis, we have $[\overline{\sigma_{arg}}/\overline{x}, \sigma_{obj}/\texttt{this}]\overline{e_{arg1}} \equiv [\overline{\sigma_{arg}}/\overline{x}, \sigma_{obj}/\texttt{this}]\overline{e_{arg2}}$, so $[\overline{\sigma_{arg}}/\overline{x}, \sigma_{obj}/\texttt{this}]\texttt{new}\, C(\overline{e_{arg1}})$ $\equiv [\overline{\sigma_{arg}}/\overline{x}, \sigma_{obj}/\texttt{this}]\texttt{new}\, C(\overline{e_{arg2}})$.

□

Lemma 11. *The evaluation of a method invocation and its specialization are equivalent.*

If $\vdash P^G = \overline{CD^G}\, e_m :^G \diamond$ and $\overline{CD^G}, \varnothing \vdash e_m :^G T^G_m$ and $P^G, \varnothing, \varnothing \vdash \sigma_{obj}.m\_n(\overline{\sigma_{arg}}) \Rightarrow \langle e', \overline{CD^S} \rangle$ and $\overline{CD^G} \vdash \sigma_{obj}.m(\overline{\sigma_{arg}}) \longrightarrow^G e_1$ and $\overline{CD^S} \vdash e' \longrightarrow^S e_2$, then $e_1 \equiv e_2$.

Proof. By induction on $\longrightarrow^G$ and the implementations of *specializem* and *specializemn.*

The only $\longrightarrow^G$ applicable in the hypothesis is R-INV^G because the implicit object and the arguments are all values, and the method invocation expression is not evaluated to an error. By SP-INVC, we have that $e' = \sigma_{obj}.m\_n(\overline{\sigma_{arg}})$. Then, assuming that the method bodies of m and m_n are e_m and e_{m_n}, respectively, we have to prove that $[\overline{\sigma_{arg}}/\overline{x}, \sigma_{obj}/\texttt{this}]e_m \equiv [\overline{\sigma_{arg}}/\overline{x}, \sigma_{obj}/\texttt{this}]e_{m\_n}$. Since Lemma 10 proves that substitution preserves equivalence, we have to prove that $e_m \equiv e_{m\_n}$. Then, we analyze the m_n specialized method created from m by the *specialize* function defined in Appendix J, to see if the body of all the specialized methods are equivalent to the original one (i.e., $e_m \equiv e_{m\_n}$).

If the method has not been specialized yet, *specializem* calls *specializemn*, which creates a new m_n method. The body of m_n is the specialization of the body of the original m method. By Lemma 9, we have that both bodies are equivalent. If the m method has already been specialized, *specializem* returns the existing one (previous case). The last implementation of *specializem* returns a non-existent method when m is not in P^G. However, this cannot happen since the method invocation is not evaluated to an error.

Besides the implementation discussed above, *specializemn* has another implementation for specializing an already specialized method. In that case, it creates a new method with the same body as the existing one (previous case). The last implementation of *specializemn* also returns a method with the specialization of the body of the original method (as in the previous case). The only difference is that its type is changed (`Object` replaces `dynamic`), but that fact does not change that $e_m \equiv e_{m\_n}$, by the definition of $\equiv$.

□

Lemma 12. *One-step equivalence of specialization.*

If $\vdash P^G = \overline{CD^G}\, e_1 :^G \diamond$ and $\overline{CD^G}, \varnothing \vdash e_1 :^G T_1^G$ and $P^G, \varnothing, \varnothing \vdash e_1 \Rightarrow \langle e_2, \overline{CD^S}\rangle$, then

- $e_1 \in$ Values and $e_1 = e_2$, or
- $\overline{CD^G} \vdash e_1 \longrightarrow^G error$ and $\overline{CD^G} \vdash e_2 \longrightarrow^{SE} error$, or
- $\exists\, e_3\, e_4\, .\, \overline{CD^G} \vdash e_1 \longrightarrow^G e_3$ and $\overline{CD^S} \vdash e_2 \longrightarrow^S e_4$ and $e_3 \equiv e_4$

Proof. By induction on the typing derivation of FC#$^{\mathrm{G}}$, analyzing the 3 possible conclusions of the lemma.

- Cases T-Var$^{\mathrm{G}}$ and T-This$^{\mathrm{G}}$. These cases will not occur, since Γ is empty in the hypothesis.
- Case T-FieldC$^{\mathrm{G}}$. $\quad e_1 = e.f \qquad \overline{CD^G}, \Gamma \vdash e.f :^G T^G \qquad \overline{CD^G}, \Gamma \vdash e :^G C$

  The specialization rule for this case is SP-Field. By definition of values, e_1 is not a value; and neither is e_2, by definition of T-FieldC$^{\mathrm{G}}$ and SP-Field.

  If $\overline{CD^G} \vdash e_1 \longrightarrow^G error$, the R-FieldE$^{\mathrm{G}}$ rule is the only one that reduces field access to error. As proved in Appendix D, T-FieldC$^{\mathrm{G}}$ and R-FieldE$^{\mathrm{G}}$ cannot occur over the same e_1, so that case is not applicable either.

  For the last alternative, applying the induction hypothesis to e, we have 3 options: 1) if e and its specialization e' are the same value, then $e.f$ and $e'.f$ will be reduced to the same expression ($e_3 = e_4$ so $e_3 \equiv e_4$), because R-Field$^{\mathrm{G}}$ and R-Field$^{\mathrm{S}}$ perform the same evaluation; 2) e cannot be evaluated to an error, since e_1 is typed by T-FieldC$^{\mathrm{G}}$; and 3) if e can be reduced to e_5 and its specialization e' is evaluated to e_6, then $e_5 \equiv e_6$. In that case, by definition of $\equiv$, we have $e_5.f = e_3 \equiv e_6.f = e_4$.
- Case T-FieldD$^{\mathrm{G}}$. $\quad e_1 = e.f \qquad \overline{CD^G}, \Gamma \vdash e.f :^G dynamic \qquad \overline{CD^G}, \Gamma \vdash e :^G dynamic$

  As in the previous case, e_1 and e_2 cannot be values.

  If e_1 is evaluated to an error, that is because e is a value σ and it does not provide the f field (R-FieldE$^{\mathrm{G}}$). By the induction hypothesis, the specialization of e is also σ, so $\sigma.f$ will produce an error by R-FieldE$^{\mathrm{S}}$.

  For the last alternative, 1) and 3) are proved as in the previous case. For 2), we have that $\overline{CD^G} \vdash e \longrightarrow^G error$, so $\overline{CD^G} \vdash e_1 \longrightarrow^G e_3$ cannot occur; i.e., 2) is not a valid alternative.

- Case T-InvC$^{\text{G}}$. $\quad e_1 = e_{obj}.m(\overline{e_{arg}}) \qquad \overline{CD^G}, \Gamma \vdash e_1 :^G T_r^G \qquad \overline{CD^G}, \Gamma \vdash e_{obj} :^G C_{obj}$

  e_1 is not a value (first case) by definition of values.

  If e_1 is reduced to an error, it must be by applying the rules T-InvE$^{\text{G}}$, T-ParE$^{\text{G}}$ or T-ArgE$^{\text{G}}$. The three rules require premises that cannot be fulfilled because they are contrary to the hypothesis (premises of T-InvC$^{\text{G}}$). Therefore, e_1 cannot be reduced to an error.

  Since the two previous cases are not applicable, we have to prove the third case; i.e., $\exists\, e_3\, e_4 \,.\, \overline{CD^G} \vdash e_1 \longrightarrow^G e_3$ and $\overline{CD^S} \vdash e_2 \longrightarrow^S e_4$ and $e_3 \equiv e_4$. If we analyze the specialization rules, we see that SP-InvE, SP-InvC and SP-InvU can be applied. If SP-InvE is applied, $e_1 = e_2$ and hence $e_3 = e_4$. In the two other cases, we have that $e_2 = e'_{obj}.m\_n(\overline{e'_{arg}})$.

  Now, we analyze the evaluation rules. If R-Inv$^{\text{G}}$ is evaluated, then e_{obj} and $\overline{e_{arg}}$ are all values, so Lemma 11 tells us that $e_3 \equiv e_4$. If R-InvC$^{\text{G}}$ is evaluated, e_{obj} is evaluated to e_{obj2} and its specialization e'_{obj} is evaluated to e'_{obj2}. By the induction hypothesis, we know that they are equivalent ($e_{obj2} \equiv e'_{obj2}$). Then, since $e_{obj2} \equiv e'_{obj2}$ and $\overline{e_{arg}} \equiv \overline{e'_{arg}}$, we have that $e_{obj2}.m(\overline{e_{arg}}) = e_3 \equiv e'_{obj2}.m\_n(\overline{e_{arg}})$ by definition of $\equiv$. The R-InvAC$^{\text{G}}$ rule is proved the same way.

- Case T-InvD$^{\text{G}}$. $\quad e_1 = e_{obj}.m(\overline{e_{arg}}) \qquad \overline{CD^G}, \Gamma \vdash e_1 :^G \mathit{dynamic}$

  $\overline{CD^G}, \Gamma \vdash e_{obj} :^G \mathit{dynamic}$

  Due to its syntax, e_1 is not a value.

  If e_1 is evaluated to an error, that is because e_{obj} is a value and it does not provide the m method (R-InvE$^{\text{G}}$), or does not provide the correct number of arguments (R-ParE$^{\text{G}}$), or one of the types is not correct (R-ArgE$^{\text{G}}$). For any of those cases, SP-InvE would specialize the method to the original expression, producing the same runtime error by R-InvE$^{\text{S}}$, R-ParE$^{\text{S}}$ and R-ArgE$^{\text{S}}$, respectively.

  For the third alternative ($\exists\, e_3\, e_4 \,.\, \overline{CD^G} \vdash e_1 \longrightarrow^G e_3$ and $\overline{CD^S} \vdash e_2 \longrightarrow^S e_4$ and $e_3 \equiv e_4$), the proof is the same as in the previous case.

- Case T-New$^{\text{G}}$. $\quad e_1 = \texttt{new}\ C(\overline{e_{arg}})$

  If e_1 is a value, so are $\overline{e_{arg}}$. Then, by definition Lemma 9, the specialization of $\overline{e_{arg}}$ ($\overline{e'_{arg}}$) is equivalent to $\overline{e_{arg}}$ ($\overline{e_{arg}} \equiv \overline{e'_{arg}}$). Since $\overline{e_{arg}}$ are values and $\overline{e_{arg}} \equiv \overline{e'_{arg}}$, then $\overline{e_{arg}} = \overline{e'_{arg}}$ by definition of $\equiv$. Therefore, $e_1 = \texttt{new}\, C(\overline{e_{arg}}) = e_2 = \texttt{new}\, C(\overline{e'_{arg}})$.

  e_1 cannot be evaluated to an error, since no such rule is defined in the semantics of FC#$^{\text{G}}$.

  For the third alternative, $\exists\, e_3\, e_4 \,.\, \overline{CD^G} \vdash e_1 \longrightarrow^G e_3$ and $\overline{CD^S} \vdash e_2 \longrightarrow^S e_4$ and $e_3 \equiv e_4$, the only evaluation rule to be applied is R-NewAC$^{\text{G}}$. Thus, $e_1 = \texttt{new}\, C(\overline{e_{arg}})$ and $e_2 = \texttt{new}\, C(\overline{e'_{arg}})$, being $\overline{e'_{arg}}$ the specialization of $\overline{e_{arg}}$. By Lemma 9, $\overline{e'_{arg}} \equiv \overline{e_{arg}}$, so, by definition of $\equiv$, $e_1 = \texttt{new}\, C(\overline{e_{arg}}) \equiv e_2 = \texttt{new}\, C(\overline{e'_{arg}})$.

$\square$

Property 6. *Program specialization preservers semantics (i.e., the original program and its specialization are semantically equivalent).*

If $\vdash P^G = \overline{CD^G}\, e_1 :^G \diamond$ and $\overline{CD^G}, \varnothing \vdash e_1 :^G T_1^G$ and $P^G, \varnothing, \varnothing \vdash e_1 \Rightarrow \langle e_2, \overline{CD^S}\rangle$, then

- $\exists\, \sigma\, .\, \overline{CD^G} \vdash e_1 \longrightarrow^{G*} \sigma$ and $\overline{CD^S} \vdash e_2 \longrightarrow^{S*} \sigma$, or
- $\overline{CD^G} \vdash e_1 \longrightarrow^{G*} \mathit{error}$ and $\overline{CD^S} \vdash e_2 \longrightarrow^{SE*} \mathit{error}$, or
- $\exists\, e_3\, e_4\, .\, \overline{CD^G} \vdash e_1 \longrightarrow^{G} e_3$ and $\overline{CD^S} \vdash e_2 \longrightarrow^{S} e_4$

Proof. By the consecutive application of the progress property in both languages (Properties 2 and 3) and Lemma 12 (one-step equivalence of specialization).

By Property 2, e_1 can be evaluated to a value. By Lemma 12, e_2 must be evaluated to the same value. Property 2 also allows e_1 to be evaluated to an error. Then Lemma 12 can be applied to deduce that e_2 will also be evaluated to an error. Similarly, Property 2 allows e_1 to be evaluated to another expression, the same as e_2, by Lemma 12 (we also know that both evaluations are equivalent). □

Appendix N. Additional code generation rules for FC#$^{\mathbf{G}}$

$[\![P^G = \overline{CD^G}\, e]\!](\Gamma) =$
    $[\![\overline{CD^G}]\!](\overline{CD^G}, \Gamma)$
    $[\![e]\!](\overline{CD^G}, \Gamma)$

```
    internal static class Reflection {
        internal static dynamic GetField(dynamic obj, string field) {
            return obj.GetType().GetField(field).GetValue(obj);
        }
        internal static dynamic Invoke(dynamic obj, string method,
                                        object[] args) {
            return obj.GetType().GetMethod(method).Invoke(obj, args);
        }
    }
```

$[\![CD^G = \texttt{class}\, C : C_{\mathit{super}}\, \texttt{\{}\, \overline{T_f^G\, f}\texttt{;}\; K^G\, \overline{M^G}\, \texttt{\}}]\!](\overline{CD^G}, \Gamma) =$
    `public class` $C : C_{\mathit{super}}$ `{`
        $[\![\overline{T^G\, f}]\!]$
        $[\![K^G]\!](\overline{CD^G})$
        $[\![\overline{M^G}]\!](\overline{CD^G}, \Gamma, C)$
    `}`

$[\![T^G\, f]\!] =$ `public` $|T^G|\; f$`;`

$$\dfrac{\overline{CD^G} = CD_1^G \ldots \texttt{class}\, C : C_{\mathit{super}}\, \texttt{\{} \ldots \texttt{\}} \ldots CD_n^G \qquad \overline{CD^G} = CD_1^G \ldots \texttt{class}\, C_{\mathit{super}} : C_{ss}\, \texttt{\{}\, \overline{T_{sf}^G\, f_s}\texttt{;}\; K^G\, \overline{M^G}\, \texttt{\}} \ldots CD_n^G \qquad K^G = C_{\mathit{super}}(\overline{T_{skp}^G\, f_s})\texttt{:base(} \ldots \texttt{)\{} \ldots \texttt{\}}}{\mathit{fields}(\overline{CD^G}, C) = \overline{T_{f1}^G\, f_1}, \overline{T_{f2}^G\, f_2}}$$

$[\![K^G = C(\overline{T_{p1}^G\, f_1}, \overline{T_{p2}^G\, f_2})\texttt{:base(}\overline{f_1}\texttt{)}\;\texttt{\{}\overline{\texttt{this.}f_2 = f_2\texttt{;}}\texttt{\}}]\!](\overline{CD^G}) =$
    `public` $C(\overline{|T_{p1}^G|\, f_1}, \overline{|T_{p2}^G|\, f_2})$`:` `base(`$\overline{\mathit{cast}(T_{p2}^G, T_{skp}^G) f_1}$`) {`

$$\overline{\texttt{this}.f_2 = cast(T^G_{p2}, T^G_2)\texttt{(}f_2\texttt{)};}$$
$$\texttt{\}}$$

$$\frac{\Gamma' = \overline{x}{:}\overline{T^G_p}, \texttt{this}{:}C, \Gamma \qquad \overline{CD^G}, \Gamma' \vdash e :^G T^G_e}{\begin{array}{l}[\![M^G = T^G_r\ m(\overline{T^G_p x})\ \texttt{\{return}\ e\texttt{;\}}]\!](\overline{CD^G}, \Gamma, C) = \\ \quad \texttt{public}\ |T^G_r|\ m(\overline{|T^G_p|\ x})\ \texttt{\{} \\ \qquad \texttt{return}\ cast(T^G_e, T^G_r)[\![e]\!](\overline{CD^G}, \Gamma)\texttt{;} \\ \quad \texttt{\}}\end{array}}$$

$$[\![x]\!] = \texttt{x} \qquad [\![\texttt{this}]\!] = \texttt{this} \qquad \frac{\overline{CD^G}, \Gamma \vdash e :^G C}{[\![e.f]\!](\overline{CD^G}, \Gamma) = [\![e]\!](\overline{CD^G}, \Gamma)\texttt{.f}}$$

$$\frac{\overline{CD^G}, \Gamma \vdash \overline{e_{arg}} :^G \overline{T^G_{arg}} \qquad \overline{CD^G} = CD^G_1 \ldots \texttt{class}\ C : C_{super}\ \texttt{\{}\overline{T^G_f f}\texttt{;}\ K^G\ \overline{M^G}\ \texttt{\}} \ldots CD^G_n \qquad K^G = C(\overline{T^G_p\ f_p})\texttt{:base(}\ldots\texttt{)\{}\ldots\texttt{\}}}{[\![\texttt{new}\ C(\overline{e_{arg}})]\!](\overline{CD^G}, \Gamma) = \texttt{new}\ C\,(\overline{cast(T^G_{arg}, T^G_p)[\![e_{arg}]\!](\overline{CD^G}, \Gamma)})}$$

Appendix O. Additional code generation rules for FC#^{S}

$$\begin{array}{l}[\![P^S = \overline{CD^S}\ e]\!](\Gamma) = \\ \quad [\![\overline{CD^S}]\!](\overline{CD^S}, \Gamma) \\ \quad [\![e]\!](\overline{CD^S}, \Gamma)\end{array}$$

$$\begin{array}{l}[\![CD^S = \texttt{class}\ C : C_{super}\ \texttt{\{}\overline{T^S_f\ f}\texttt{;}\ K^S\ \overline{M^S}\ \texttt{\}}]\!](\overline{CD^S}, \Gamma) = \\ \quad \texttt{public class}\ C : C_{super}\ \texttt{\{} \\ \qquad \texttt{private static object \_temp;} \\ \qquad \overline{[\![T^S\ f]\!]} \\ \qquad [\![K^S]\!](\overline{CD^S}) \\ \qquad \overline{[\![M^S]\!]}(\overline{CD^S}, \Gamma, C) \\ \quad \texttt{\}} \\ [\![T^S f]\!] = \texttt{public}\ |T^S|\ f\texttt{;}\end{array}$$

$$\frac{\begin{array}{c} fields(\overline{CD^S}, C) = \overline{T^S_{f1}\ f_1}, \overline{T^S_{f2}\ f_2} \qquad \overline{CD^S} = CD^S_1 \ldots \texttt{class}\ C : C_{super}\ \texttt{\{}\ldots\texttt{\}} \ldots CD^S_n \\ \overline{CD^S} = CD^S_1 \ldots \texttt{class}\ C_{super} : C_{ss}\ \texttt{\{}\overline{T^S_{sf}\ f_s}\texttt{;}\ K^S\ \overline{M^S}\ \texttt{\}} \ldots CD^S_n \\ K^S = C_{super}(\overline{T^S_{skp}\ f_s})\texttt{:base(}\ldots\texttt{)\{}\ldots\texttt{\}}\end{array}}{\begin{array}{l}[\![K^S = C(\overline{T^S_{p1}\ f_1}, \overline{T^S_{p2}\ f_2})\texttt{:base(}\overline{f_1}\texttt{)}\ \texttt{\{}\overline{\texttt{this.}f_2 = f_2}\texttt{;\}}]\!](\overline{CD^S}) = \\ \quad \texttt{public}\ C(\overline{|T^S_{p1}|\ f_1}, \overline{|T^S_{p2}|\ f_2})\texttt{:}\ \texttt{base}(\overline{cast(T^S_{p2}, T^S_{skp}) f_1})\ \texttt{\{} \\ \qquad \overline{\texttt{this.}f_2 = cast(T^S_{p2}, T^S_2)\texttt{(}f_2\texttt{)};} \\ \quad \texttt{\}}\end{array}}$$

$$\frac{\Gamma' = \overline{x{:}T_p^S}, \mathtt{this}{:}C, \Gamma \qquad \overline{CD^S}, \Gamma' \vdash e :^S T_e^S}{[\![M^S = T_r^S\, m(\overline{T_p^S x})\,\mathtt{\{return}\, e\mathtt{;\}}]\!](\overline{CD^S}, \Gamma, C) =}$$

```
    public |T_r^S| m(|T_p^S| x) {
        return cast(T_e^S, T_r^S)⟦e⟧(CD^S, Γ);
    }
```

$$[\![x]\!] = \mathtt{x} \qquad [\![\mathtt{this}]\!] = \mathtt{this} \qquad \frac{\overline{CD^S}, \Gamma \vdash e :^S C}{[\![e.f]\!](\overline{CD^S}, \Gamma) = [\![e]\!](\overline{CD^S}, \Gamma)\mathtt{.f}}$$

$$\frac{\overline{CD^S}, \Gamma \vdash e :^S C \quad \mathit{type}(\mathit{method}(\overline{CD^S}, C, m)) = \overline{T_p^S} \to T_r^S \qquad \overline{CD^S}, \Gamma \vdash \overline{e_{arg}} :^S \overline{T_{arg}^S}}{[\![e.m(\overline{e_{arg}})]\!](\overline{CD^S}, \Gamma) = [\![e]\!](\overline{CD^S}, \Gamma)\mathtt{.m(}\overline{\mathit{cast}(T_{arg}^S, T_p^S)[\![e_{arg}]\!](\overline{CD^S}, \Gamma)}\mathtt{)}}$$

$$\frac{\overline{CD^S}, \Gamma \vdash \overline{e_{arg}} :^S \overline{T_{arg}^S} \qquad \begin{array}{c}\overline{CD^S} = CD_1^S \ldots \mathtt{class}\, C : C_{super}\, \mathtt{\{}\overline{T_f^S f}\mathtt{;}\, K^S\, \overline{M^S}\,\mathtt{\}} \ldots CD_n^S \\ K^S = C(\overline{T_p^S\, f_p})\mathtt{:base(}\ldots\mathtt{)\{}\ldots\mathtt{\}}\end{array}}{[\![\mathtt{new}\, C(\overline{e_{arg}})]\!](\overline{CD^S}, \Gamma) = \mathtt{new}\, C\,\mathtt{(}\overline{\mathit{cast}(T_{arg}^S, T_p^S)[\![e_{arg}]\!](\overline{CD^S}, \Gamma)}\mathtt{)}}$$