A formal framework for compositional compilation

Davide Ancona* and Elena Zucca
DISI, Università di Genova,
Italy
**E-mail: davide@disi.unige.it*
www.disi.unige.it

We define a general framework for *compositional compilation*, meant as the ability of building an executable application by separate compilation and linking of single fragments, opposed to *global compilation* of the complete source application code. More precisely, compilation of a source code fragment in isolation generates a corresponding binary fragment equipped with type information, formally modeled as a *typing*, allowing type safe linking of fragments without re-inspecting code.

We formally define a notion of soundness and completeness of compositional compilation w.r.t. global compilation, and show how linking can be in practice expressed by an *entailment* relation on typings. Then, we provide a sufficient condition on such entailment to ensure soundness and completeness of compositional compilation, and compare this condition with the *principal typings* property. Furthermore, we show that this entailment relation can often be modularly expressed by an entailment relation on type environments and a subtyping relation.

We illustrate the generality of our approach by instantiating the framework on three main examples: simply typed lambda calculus, where the problem of compositional compilation reduces to compositional type inference; Featherweight Java, where the generated binary code depends on the compilation context; and an extension of Featherweight Java with a boxing/unboxing mechanism, to illustrate how the framework can also support more sophisticated forms of linking-time binary code specialization.

Keywords: Type systems; Separate compilation.

1. Introduction

Originally, compilation was *global*, meaning that the complete source code of an application was needed for performing typechecking and binary code generation. In strongly typed languages execution of a globally compiled application is guaranteed to be safe.

However, modern software systems, besides allowing global compilation,

support also *separate compilation*: open code fragments can be compiled in isolation, provided that type information on the needed external definitions are available in some form, and can then be linked in a type safe way. In this way, *smart recompilation* (also known as *selective recompilation*)[1–4] is promoted, since a fragment can be reused in many different applications, without re-inspecting its source code, provided that mutual compatibility requirements among fragments are verified. More importantly, separate compilation is essential for developing applications in modern systems where fragments can be distributed and loaded/linked dynamically, as happens in Java and C#.

Despite of this, there has been little work on formal models for separate compilation and linking, except for the seminal paper,[5] which formalized *intra-checking* (separate compilation simplified to typechecking of fragments in isolation) and *inter-checking* of *linksets* (collection of successfully compiled fragments equipped with mutual type assumptions) on simply typed lambda-calculus, and some other papers,[6,7] where Cardelli's ideas are further developed, by formally defining *compositional compilation* for Java-like languages as the combination of separate compilation and linking steps. Our approach here, instead, is to abstract away from language-specific problems, aiming at providing an abstract framework in order to capture the main general properties needed by compilation to be compositional.

To this end, we extend the simple framework in[5] in many respects:

- We define an abstract framework which can be instantiated on an arbitrary type system providing some ingredients. In particular, we abstract from the form of type assumptions, rather than just considering assumptions $x : \tau$ meaning that the fragment named x should have type τ.
- We model *linking* rather than just inter-checking, that is, a well-formed linkset is obtained by not only checking that mutual assumptions are satisfied, but also by a non trivial manipulation of the original fragments. That is, the framework supports *code specialization*.
- We expect linking to satisfy some properties which ensure that it can actually replace global compilation. First, it should be at least *sound*, in the sense that, if linking succeeds for some linkset, then we can be sure that compiling altogether the fragments we would succeed as well and get the same binaries. Furthermore, if linking is *complete*, then when it fails we can conclude that we would get

a failure as well by compiling altogether the fragments.

We show that compositional compilation is sound and complete if linking can be expressed by an *entailment* relation on typings such that each composable fragment has a typing which entails all and only others, and separate compilation always produces typings of this kind. Indeed, in this way failure of linking cannot be caused by the fact that for some fragment we have selected a typing which is not adequate in that particular linkset. We formally compare this property with the *principal typings* property,[8] formalized in a system-independent way by Wells,[9] showing that the latter notion is a specialization of the former which is obtained when the entailment relation is sound, that is, preserves validity for all typings (not just those principal). In this case, linking satisfies an extended soundness property, that is, is sound for all linksets, not only those obtained by separate compilation.

Moreover, we show that an entailment relation on typings can often be modularly expressed by an entailment relation on type environments and a subtyping relation which should satisfy in turn appropriate conditions of soundness and completeness.

We show the expressive power of the framework by outlining three rather different instantiations. First, we consider simply typed lambda calculus, as representative of languages where the type of fragments depend on the context and, hence, the hardest part of compositional compilation is type inference and code generation can be disregarded,

Then, we consider Featherweight Java, as representative of languages, such as Java and C#, where the type of fragments is fixed a priori (because of type annotations) but the corresponding generated binary fragments depend on the context, hence separate code generation is hard to achieve. We re-cast the solution to Java compositional compilation based on the notion of polymorphic bytecode[7] into our abstract framework, showing in particular that linking implies in this case specialization of binary code, via instantiation of type variables. Finally, we extend Featherweight Java with a boxing/unboxing mechanism, to illustrate how the framework also supports code specialization in a broader sense, that is, linking implies that conditional code is transformed in one of the branches.

The rest of the paper is organized in two sections, the former devoted to the formal definition of the framework and the latter to the instantiation examples, and the Conclusion.

In Sect.2 we first define compositional compilation on top of a generic type system providing some basic ingredients, and formally define sound-

ness and completeness w.r.t. global compilation (Sect.2.1); then we introduce the more concrete notion of linking by entailment on typings (Sect.2.2), and investigate the relation with the notion of principal typing[8,9] (Sect.2.3); finally, we introduce the even more concrete notion of linking by entailment on environments and subtyping (Sect.2.4).

In Sect.3 we present instantiations of the framework on simply typed lambda calculus (Sect.3.1), Featherweight Java (Sect.3.2), and Featherweight Java extended with a boxing/unboxing mechanism (Sect.3.3).

In the Conclusion we summarize the contribution of the paper and outline further research directions.

2. Formalizing compositional compilation

In this section, we first define, in Sect.2.1, a schema formalizing both global and compositional compilation on top of a given type system $\mathcal{T}$. In global compilation a collection of source fragments can be compiled only if self-contained and altogether. In compositional compilation, instead, any single source fragment can be compiled in isolation under type assumptions on missing entities; then, successfully compiled fragments can be linked together if their mutual assumptions are satisfied. We formally express a property of soundness and completeness of compositional compilation w.r.t. global compilation.

In the framework of Sect.2.1, linking is modeled in a very abstract way as a judgment $\vdash_C [x_i \triangleright \Gamma_i \vdash \tau_i{}^{i\in 1..n}] : \mathtt{T}$ asserting that a collection of (named by x_i) binary fragments τ_i, equipped with type assumptions Γ_i, can be successfully linked together producing new binaries $\mathtt{T}$. In Sect.2.2, we introduce a more concrete model where linking is expressed by an *entailment* relation on pairs $\Gamma \vdash \tau$ (*typings*), and in Sect.2.3 we formally compare this notion with the semantic relation on typings (a typing is stronger than another iff it is a typing of less terms) which is used to state the *principal typings* property.[8,9]

Finally, in Sect.2.4 we provide an even more concrete view of linking, where the relation on typings is defined in terms of an entailment relation on type environments Γ and a subtyping relation on types depending on a given type environment $\Gamma \vdash \tau <: \#\tau'$.

2.1. *An abstract framework*

We start by listing the basic syntactic categories and judgments the type system $\mathcal{T}$ should define.

Source fragments (Terms) $\mathtt{s}$ defined over a given numerable infinite set

of variables $x \in \mathit{Var}$.

Binary fragments (Types) τ. Note that, since our notion of typechecking also includes binary code generation, types correspond to binary fragments.

Type assumptions γ. They always include the type assumptions of the form $x{:}\tau$, but possibly also other kinds as shown in Sect.3.

Type environments Γ. A type environment is just a possibly empty sequence of type assumptions. The term Γ, Γ' denotes the concatenation of Γ and Γ', when it is defined (because the concatenation operator might be partial), and the empty type environment is denoted by $\emptyset$.

Type judgments $\Gamma{\vdash}\mathtt{s}{:}\tau$ with the following meaning: term $\mathtt{s}$ has type τ in the type environment Γ.

Binary fragments with assumptions (Typings) $t = \Gamma{\vdash}\tau \in \mathit{Typing}$. Note that, analogously to what happens for types, typings correspond to binary fragments equipped with type assumptions. If $\Gamma{\vdash}\mathtt{s}{:}\tau$ is a valid judgment, then we say that $\Gamma{\vdash}\tau$ is a typing of $\mathtt{s}$; we denote by $\mathit{typings}(\mathtt{s})$ the set of all the typings of a term $\mathtt{s}$, and by $\mathit{terms}(t)$ the set of all terms which have typing t. A term $\mathtt{s}$ is typable if $\mathit{typings}(\mathtt{s}) \neq \emptyset$.

The type system $\mathcal{T}$ consisting of the ingredients above formalizes the well-formedness rules of terms, which are only language-dependent and do not take into account any compilation mechanism. On top of $\mathcal{T}$ we can model two different approaches to compilation, which we will call *global* and *compositional*, respectively.

Let $\mathtt{S}$ and $\mathtt{T}$ denote collections of distinctly named source and binary fragments, respectively, that is,

$$\mathtt{S} = [x_i{\triangleright}\mathtt{s}_i^{i\in 1..n}] \qquad \mathtt{T} = [x_i{\triangleright}\tau_i^{i\in 1..n}]$$

where $x_i^{i\in 1..n}$ are all distinct.

In the first approach, a collection of named source fragments can be compiled only if it is self-contained. Global compilation is formalized by the judgment:

Global compilation $\vdash_G \mathtt{S}{:}\mathtt{T}$

which is defined by the rule in Fig.1, which states that a collection of named terms can be compiled only if all terms typecheck in the context of full type information about each other.

$$(glob)\frac{x_i : \tau_i^{i\in 1..n} \vdash \mathtt{s}_j{:}\tau_j, j \in 1..n}{\vdash_G [x_i \triangleright \mathtt{s}_i^{i\in 1..n}]{:}[x_i \triangleright \tau_i^{i\in 1..n}]}$$

Fig. 1. Global compilation

Note that global compilation is in practice expected to be a *function* from sources into binaries; however, this requirement is never needed in the following technical treatment, hence we prefer to keep the more abstract view of a relation.

In the second approach, instead, single source fragments can be compiled in isolation, generating binary fragments which are equipped with type assumptions on the missing entities, and which can then be linked together provided that mutual assumptions are satisfied. A compositional compilation mechanism is defined by two new ingredients: *separate compilation* and *linking*. Separate compilation is formalized by selecting a subset of the valid type judgments:

Separate compilation $\Gamma \vdash_C \mathtt{s}{:}\tau$ (s.t. if $\Gamma \vdash_C \mathtt{s}{:}\tau$ holds, then $\Gamma \vdash \mathtt{s}{:}\tau$ holds)

This models the fact that when we compile a term in isolation, we need to fix type assumptions on missing entities. As already noted for global compilation, also separate compilation is in practice expected to be a function from sources into typings[a]; however, again this requirement is not needed for the following technical treatment, hence we prefer to keep the more abstract view of a relation (a subset of the valid judgments).

Linking is performed on a *linkset*, that is, a collection of distinctly named typings:

Linksets $\mathtt{L} = [x_i \triangleright \Gamma_i \vdash \tau_i{}^{i\in 1..n}]$ where $x_i^{i\in 1..n}$ are all distinct,

and is formalized by the judgment:

Linking $\vdash_C \mathtt{L}{:}\mathtt{T}$

This judgment models an effective procedure for linking together a linkset obtained by separate compilation of a collection of terms. Note that this check does *not* depend on the source code.

[a]Type assumptions can be extracted from source code either trivially, if they are specified by the programmer, or by an inference algorithm; we do not care here about this distinction.

Compositional compilation can then be formalized by the judgment:

Compositional compilation $\vdash_C \mathtt{S}{:}\mathtt{T}$

which is defined by the rule in Fig.2.

$$(comp)\ \frac{\Gamma_i \vdash_C \mathtt{s}_i{:}\tau_i' \quad \vdash_C [x_i \triangleright \Gamma_i \vdash \tau_i'^{\,i\in 1..n}]{:}[x_i \triangleright \tau_i^{i\in 1..n}]}{\vdash_C [x_i \triangleright \mathtt{s}_i^{i\in 1..n}]{:}[x_i \triangleright \tau_i^{i\in 1..n}]}$$

Fig. 2. Compositional compilation

The advantages of compositional compilation (that is, separate compilation plus linking) w.r.t. global compilation are clear. Each fragment can be compiled without inspecting the fragments it depends on; then, a collection of fragments can be put together without re-inspecting source code. However, in order to really offer these advantages, compositional compilation should satisfy some properties which ensure that it can actually replace global compilation. This issue was not considered in.[5]

Definition 2.1. Compositional compilation $\vdash_C$ is *sound and complete (w.r.t. global compilation)* iff for all $\mathtt{S}$, $\mathtt{T}$

$$\vdash_C \mathtt{S}{:}\mathtt{T} \iff \vdash_G \mathtt{S}{:}\mathtt{T}.$$

The $\Rightarrow$ implication corresponds to soundness of linking and, thus, can be considered as the minimal requirement to be met by a linking procedure. Indeed, it ensures that, in case of successful linking, global (re)compilation would succeed as well and generate the same binaries. On the other hand, we would like to be sure that, if linking fails, then we would get failure by global (re)compilation as well. This is expressed by the $\Leftarrow$ implication which corresponds to completeness of linking. Note that soundness and completeness of linking allow optimal selective recompilation: we *never* need to recompile a source fragment when we put it together with others, hence, even more, we never need to have source code available. Note that, even though desirable, completeness is not as fundamental as soundness. Indeed, as will be shown in the examples in Sect.3, in general linking is not complete if in separate compilation we select arbitrary judgments (e.g., all valid judgments); the intuition suggests that, in order to achieve completeness, linksets should

only contain those typings that better represent a given source. As we will see in the next sections, the intuitive notion of best representative typing is intimately connected with that of principal typing. On the other hand, soundness can be reasonably extended to all valid judgments.

2.2. *Linking by entailment*

We introduce now a more concrete model where linking is expressed by a relation $<\#$ on typings (expected to be computable). We call this relation *entailment* since we will see later that in reasonable cases it is expected to preserve typability of terms. However, in principle there are no requirements on this relation. We show that a necessary and sufficient condition for soundness and completeness of compositional compilation is that each *composable* term has a $<\#$-principal typing, where a composable term $\mathtt{s}$ is a source fragment for which there exists an $\mathtt{S}$ which contains $\mathtt{s}$ and which can be globally compiled. This means that, for each composable term $\mathtt{s}$, it is possible to select a typing t of $\mathtt{s}$ which entails all and only all the typings of $\mathtt{s}$, hence can be considered a "representative" of the typings of $\mathtt{s}$.

In the following, an *entailment relation* is a relation on *Typing*. Moreover, if $<\#$ is an entailment relation and $t<\#t'$, then we say that t *entails* t'.

Definition 2.2. A typing t of $\mathtt{s}$ is a $<\#$*-principal typing* of $\mathtt{s}$ iff it entails all and only all the typings of $\mathtt{s}$; that is, $\forall t' \in \mathit{Typing} \quad t<\#t' \Leftrightarrow \mathtt{s} \in \mathit{terms}(t')$.

A type system $\mathcal{T}$ *has* $<\#$*-principal typings* if each typable term has a $<\#$-principal typing.

Definition 2.3. A term $\mathtt{s}$ is *composable* iff there exists $\mathtt{S} = [x_i \triangleright \mathtt{s}_i^{i\in 1..n}]$ s.t.

(1) $\mathtt{s}$ is in $\mathtt{S}$, that is, there exists $k \in 1..n$ s.t. $\mathtt{s} = \mathtt{s}_k$;
(2) $\mathtt{S}$ is globally compilable, that is, there exists $\mathtt{T}$ s.t. $\vdash_G \mathtt{S}{:}\mathtt{T}$.

Note that the notion of composable is in general stricter than that of typable. Indeed, by definition of composable term and of global compilation, a composable term is always typable, whereas the converse might not hold. This happens in type systems where deciding whether the type assumptions of a given type environment are satisfiable is so hard that in practice is better to allow typings with unsatisfiable type environments, with the drawback that some type error can be detected only at linking time rather than at (separate) compilation time (for a concrete example, see FJP as defined in Sect.3.2).

Definition 2.4. Let $<\#$ be an entailment relation. We denote by $\vdash^{<\#}$ the *separate compilation and linking induced by* $<\#$, defined as follows:

- $\Gamma \vdash^{<\#} \mathtt{s}{:}\tau$ iff $\Gamma{\vdash}\tau$ is a $<\#$-principal typing of $\mathtt{s}$;
- $\vdash^{<\#}[x_i{\triangleright}\Gamma_i{\vdash}\tau_i'^{i\in 1..n}]{:}[x_i{\triangleright}\tau_i^{i\in 1..n}]$ iff
  for all $i \in 1..n, (\Gamma_i{\vdash}\tau_i'){<\#}(x_j : \tau_j{}^{j\in 1..n}{\vdash}\tau_i)$.

We denote by $\vdash_C^{<\#}$ the *compositional compilation induced by* $<\#$, defined by the rule *(comp)* in Fig.2 on top of the separate compilation and linking $\vdash^{<\#}$.

Theorem 2.1. *Let $<\#$ be an entailment relation. Then, the compositional compilation induced by $<\#$ is sound and complete iff each composable term has a $<\#$-principal typing.*

Proof.

$\Rightarrow$ Let $\mathtt{s}$ be a composable term, then by definition there exists $\mathtt{S} = [x_i{\triangleright}\mathtt{s}_i^{i\in 1..n}]$ s.t.

(1) $\mathtt{s}$ is in $\mathtt{S}$, that is, there exists $k \in 1..n$ s.t. $\mathtt{s} = \mathtt{s}_k$;
(2) $\mathtt{S}$ is globally compilable, that is, there exists $\mathtt{T}$ s.t. $\vdash_G \mathtt{S}{:}\mathtt{T}$.

By 2 and by completeness of compositional compilation, $\vdash_C^{<\#}\mathtt{S}{:}\mathtt{T}$ is valid, and by 1 and by definition of compositional compilation induced by $<\#$, $\Gamma_k \vdash^{<\#} \mathtt{s}{:}\tau_k'$ for some Γ_k, τ_k'. Finally, by definition of separate compilation $\vdash^{<\#}$, we can conclude that $\Gamma_k{\vdash}\tau_k'$ is a $<\#$-principal typing of $\mathtt{s}$.

$\Leftarrow$ The following equivalences hold:
$\vdash_C^{<\#}[x_i{\triangleright}\mathtt{s}_i^{i\in 1..n}]{:}[x_i{\triangleright}\tau_i^{i\in 1..n}]$
$\iff$ (def. of $\vdash_C^{<\#}$)
$\Gamma_i \vdash^{<\#} \mathtt{s}_i{:}\tau_i'$ and
$\vdash^{<\#}[x_i{\triangleright}\Gamma_i{\vdash}\tau_i'^{i\in 1..n}]{:}[x_i{\triangleright}\tau_i^{i\in 1..n}]$ for all $i \in 1..n$
$\iff$ (def. of $\vdash^{<\#}$)
$\Gamma_i{\vdash}\tau_i'$ is a $<\#$-principal typing of $\mathtt{s}_i$ and
$(\Gamma_i{\vdash}\tau_i'){<\#}(x_j : \tau_j{}^{j\in 1..n}{\vdash}\tau_i)$ for all $i \in 1..n$
$\iff$ (def. of $<\#$-principal typing and hypothesis)
$x_i : \tau_i^{i\in 1..n}{\vdash}\mathtt{s}_j{:}\tau_j$ and $\mathtt{s}_j$ composable for all $j \in 1..n$
$\iff$ (def. of $\vdash_G$ and of composable term)
$\vdash_G[x_i{\triangleright}\mathtt{s}_i^{i\in 1..n}]{:}[x_i{\triangleright}\tau_i^{i\in 1..n}]$ $\square$

2.3. *Compositional compilation and principal typings*

In this section we investigate the relation between the notion of $<\#$-principal typing and that of principal typing introduced, in its general formulation, by Wells,[9] which is based on a fixed semantic relation on typings (a typing is stronger than another iff it is a typing of less terms). We show that the latter notion is a specialization of the former which is obtained when the entailment relation is sound. In this case, besides soundness and completeness, compositional compilation satisfies an *extended soundness property* (Theorem 2.3) which amounts to say that linking is sound (but not necessarily complete) for all linksets, not only those which are formed by principal typings.

Definition 2.5. For all typings t, t', we say that t is *stronger* than t', and write $t \leq t'$, iff $\mathit{terms}(t) \subseteq \mathit{terms}(t')$.

A typing t is *principal* for $\mathtt{s}$ iff it is a typing of $\mathtt{s}$ which is stronger than all typings of $\mathtt{s}$. A term $\mathtt{s}$ *has a principal typing* iff there exists a typing principal for $\mathtt{s}$; conversely, a typing *is principal* iff it is principal for some $\mathtt{s}$. A type system $\mathcal{T}$ *has principal typings* if each typable term has a principal typing.

Proposition 2.1. *The following facts hold:*

(1) If a type system $\mathcal{T}$ has principal typings, then it has $<\#$-principal typings for some entailment relation $<\#$.
(2) The opposite implication does not hold.

Proof.

(1) It suffices to show that $\mathcal{T}$ has $<\#$-principal typings with $<\# = \leq$. Indeed, a principal typing t of $\mathtt{s}$ is a $\leq$-principal typing of $\mathtt{s}$ since if $t \leq t'$, then $\mathtt{s} \in \mathit{terms}(t')$ by definition of $\leq$. On the other hand, if $\mathtt{s} \in \mathit{terms}(t')$ then $t \leq t'$ since by definition of principal typing t is the minimum typing of $\mathtt{s}$ w.r.t. $\leq$.
(2) Consider example 2.1 in the following. □

Definition 2.6. An entailment relation $<\#$ is *sound* iff, for all typings t, t',

$$t <\# t' \Rightarrow \mathit{terms}(t) \subseteq \mathit{terms}(t').$$

Theorem 2.2. *A type system $\mathcal{T}$ has principal typings iff it has $<\#$-principal typings for some sound entailment relation $<\#$.*

Proof.

$\Rightarrow$ By proposition 2.1 and the fact that $\leq$ is sound by definition.

$\Leftarrow$ By contradiction, assume that the system has no principal typings, hence there exists a typable term $\mathtt{s}$ which has no principal typing. However, $\mathtt{s}$ has a $<\#$-principal typing t. Since $\mathtt{s}$ has no principal typing, there exists t' typing of $\mathtt{s}$ s.t. t is not stronger than t'. This means, by definition of $\leq$, that there exists a term $\mathtt{s}'$ s.t. $\mathtt{s}' \in \mathit{terms}(t), \mathtt{s}' \notin \mathit{terms}(t')$. Since t' is a typing of $\mathtt{s}$, and t is $<\#$-principal typing of $\mathtt{s}$, $t<\#t'$ holds, hence $<\#$ would not be sound. □

Remark 2.1. Note that if a type system $\mathcal{T}$ has $<\#$-principal typings for a sound entailment relation $<\#$, such relation does not necessarily coincide with $\leq$: indeed, by soundness trivially $<\# \subseteq \leq$, but the converse inclusion does not hold in general. For instance, if there exists a term with two different principal typings t and t', then $\mathcal{T}$ has $<\#$-principal typings for the sound entailment $<\#$ defined by $\leq \setminus \{(t', t)\} \subsetneq \leq$.

Theorem 2.3. *Let $<\#$ be a sound entailment relation and $\vdash_C$ the separate compilation defined by*

$$\Gamma \vdash_C \mathcal{S} : \tau \iff \Gamma \vdash \mathcal{S} : \tau.$$

Then the compositional compilation $\vdash_C$ defined by rule (comp) in Fig.2 on top of the separate compilation $\vdash_C$ and of the linking $\vdash^{<\#}$ is sound, that is, for all $\mathcal{S}, \mathcal{T}$

$$\vdash_C \mathcal{S} : \mathcal{T} \Rightarrow \vdash_G \mathcal{S} : \mathcal{T}.$$

Proof. By definition $\vdash_C [x_i \triangleright \mathtt{s}_i^{\,i \in 1..n}] : [x_i \triangleright \tau_i^{\,i \in 1..n}]$ implies $\Gamma_i \vdash \mathtt{s}_i : \tau_i'$ and $(\Gamma_i \vdash \tau_i') <\# (x_j : \tau_j{}^{j \in 1..n} \vdash \tau_i)$. Then, by soundness of $<\#$ we deduce $x_j : \tau_j{}^{j \in 1..n} \vdash \mathtt{s}_i : \tau_i$ for $i \in 1..n$ and, therefore, conclude by definition of global compilation. □

Example 2.1. Given a $\mathcal{T}$ which does not have principal typings, we show how it is possible to define a relation $<\#$ so that each typable term $\mathtt{s}$ has a $<\#$-principal typing.

Let us consider, for instance, system F in Curry style, which is known not to have principal typings[b]:[9]

$$\mathtt{s} ::= x \mid (\mathtt{s}_1\mathtt{s}_2) \mid (\lambda x.\mathtt{s})$$

$$\tau ::= \alpha \mid \tau_1 \to \tau_2 \mid (\forall\, \alpha.\tau)$$

Without any loss of generality, we may assume that there exists a bijection η between type and term variables. Then any λ-term can be encoded into a type in the following way:

$$\begin{array}{l} enc(x) = \eta(x) \\ enc(\mathtt{s}_1\mathtt{s}_2) = enc(\mathtt{s}_1) \to enc(\mathtt{s}_2) \\ enc(\lambda x.\mathtt{s}) = (\forall\, \eta(x).enc(\mathtt{s})) \end{array}$$

Let *dec* denote the inverse function of *enc*; then $<\#$ is the relation on F typings defined as follows:

$$\Gamma, x : \tau_1 \vdash \tau_2 <\# \Gamma' \vdash \tau' \text{ iff } \Gamma' \vdash dec(\tau_1) : \tau' \text{ is valid in F.}$$

Note that in the definition x is a meta-variable (and not a fixed variable) corresponding to the rightmost type assumption in the environment (recall that we assume that type environments are sequences of type assumptions). We claim that any typable term $\mathtt{s}$ in F has a $<\#$-principal typing. Indeed, if $\mathtt{s}$ is typable in F, then we can choose a typing $\Gamma \vdash \tau$ of $\mathtt{s}$ and derive from it another typing $t = \Gamma, x : enc(\mathtt{s}) \vdash \tau$ (where x is a term variable not defined in Γ). By weakening t is a typing of $\mathtt{s}$, and, by definition, for all t', $t <\# t'$ iff t' is a typing of $dec(enc(\mathtt{s})) = \mathtt{s}$.

As one would expect $<\#$ fails to be sound; for instance, if $t_1 = x : \forall\, \alpha.\alpha \to \alpha \vdash \forall\, \alpha.\alpha \to \alpha$ and $t_2 = \emptyset \vdash \forall\, \alpha.\alpha \to \alpha$, then $t_1 <\# t_2$, t_1 is a typing of x, but not t_2.

2.4. *Linking by environment entailment and subtyping*

In Sect.2.2 we have shown that compositional compilation can be expressed in terms of an entailment relation $<\#$ between typings, and in Sect.2.3 that such a relation is expected to be sound, and, therefore, a subset of the semantic relation $\leq$ defined in Def. 2.5.

In this section we provide a practical and general schema to define $<\#$ modularly on top of an entailment relation on type environments (which is

[b]In particular, $(x\ x)$ does not have a principal typing, hence, for sake of simplicity, here we can restrict to system F without the `let` construct.

expected to be at least sound, that is, if $\Gamma <\# \Gamma'$ is valid, then whatever holds in Γ, holds in Γ' as well), and a subtyping relation (which is expected to be at least sound, that is, if $\Gamma \vdash \tau <\# \tau'$ is valid, then under the assumptions in Γ, whatever has type τ, has type τ' as well).

To achieve that, a type system has to provide the following additional ingredients:

Environment Entailment A relation (not necessarily a pre-order) between type environments: $\Gamma_1 <\# \Gamma_2$ means that Γ_1 entails Γ_2, hence Γ_1 contains assumptions stronger than those in Γ_2.

Subtyping relation A subtyping relation (not necessarily a pre-order) defined by a judgment of the form $\Gamma \vdash \tau_1 <\# \tau_2$, meaning that τ_1 is a subtype of τ_2 in Γ.

Type variable substitution A standard notion of substitution σ which is a finite map from type variables to types[c], where $\sigma(\tau)$ and $\sigma(\Gamma)$ denote the usual capture avoiding substitution applied to a type and a type environment, respectively. We assume that terms do not contain free type variables, hence $\sigma(\mathtt{s}) = \mathtt{s}$ for all $\mathtt{s}$.
Furthermore, we assume that substitutions well-behave w.r.t. typings, that is, the following property is verified:
for all $\sigma, \Gamma, \mathtt{s}, \tau$ $\Gamma \vdash \mathtt{s}{:}\tau$ implies $\sigma(\Gamma) \vdash \mathtt{s}{:}\sigma(\tau)$

With the ingredients above, the entailment relation $<\#$ between typings can be defined by the following general rule:

$$(entail)\ \frac{\Gamma_2 <\# \sigma(\Gamma_1) \qquad \Gamma_2 \vdash \sigma(\tau_1) <\# \tau_2}{(\Gamma_1 \vdash \tau_1) <\# (\Gamma_2 \vdash \tau_2)}$$

This rule captures the intuition that $<\#$ is expected to be transitive, to be covariant w.r.t. subtyping and contravariant w.r.t. entailment between environments, and finally, to satisfy $\Gamma \vdash \tau <\# \sigma(\Gamma) \vdash \sigma(\tau)$.

The following theorems express sufficient conditions on the entailment between type environments and on subtyping to ensure that the entailment defined by *(entail)* is sound and complete.

Definition 2.7. An environment entailment $<\#$ is sound iff for all Γ, Γ' $\Gamma <\# \Gamma' \Rightarrow$ for all $\mathtt{s}, \tau$ $\Gamma' \vdash \mathtt{s}{:}\tau \Rightarrow \Gamma \vdash \mathtt{s}{:}\tau$; it is sound and complete iff
for all Γ, Γ' $\Gamma <\# \Gamma' \iff$ for all $\mathtt{s}, \tau$ $\Gamma' \vdash \mathtt{s}{:}\tau \Rightarrow \Gamma \vdash \mathtt{s}{:}\tau$.

[c] In type systems with no type variables this notion collapses to the unique empty substitution.

Definition 2.8. A subtyping relation $<\#$ is sound iff for all Γ, τ, τ' $\Gamma \vdash \tau<\#\tau' \Rightarrow$ for all $\mathtt{s}$ $\Gamma\vdash\mathtt{s}{:}\tau \Rightarrow \Gamma\vdash\mathtt{s}{:}\tau'$; it is is sound and complete iff for all Γ, τ, τ'
$\Gamma \vdash \tau<\#\tau' \iff$ for all $\mathtt{s}$ $\Gamma\vdash\mathtt{s}{:}\tau \Rightarrow \Gamma\vdash\mathtt{s}{:}\tau'$.

Theorem 2.4. *If the environment entailment and the subtyping relation are sound, then the typing entailment $<\#$ defined by rule (entail) is sound as well.*

Proof. Let us assume that $(\Gamma_1\vdash\tau_1)<\#(\Gamma_2\vdash\tau_2)$ and that $\Gamma_1\vdash\mathtt{s}{:}\tau_1$. By definition of $<\#$ and by the property of substitutions $\Gamma_2<\#\sigma(\Gamma_1)$ and $\sigma(\Gamma_1)\vdash\mathtt{s}{:}\sigma(\tau_1)$, therefore by soundness of environment entailment, $\Gamma_2\vdash\mathtt{s}{:}\sigma(\tau_1)$. By definition of $<\#$ $\Gamma_2 \vdash \sigma(\tau_1)<\#\tau_2$, therefore we can conclude by soundness of the subtyping relation that $\Gamma_2\vdash\mathtt{s}{:}\tau_2$. □

Theorem 2.5. *If the environment entailment and the subtyping relation are sound and complete, if all composable terms have a principal typing $\Gamma_1\vdash\tau_1$ s.t. for all typings $\Gamma_2\vdash\tau_2$ if $\Gamma_1\vdash\tau_1 \leq \Gamma_2\vdash\tau_2$, then there exists a substitution σ s.t.*

(1) for all $\mathtt{s}, \tau$ if $\sigma(\Gamma_1)\vdash\mathtt{s}{:}\tau$, then $\Gamma_2\vdash\mathtt{s}{:}\tau$;
(2) for all $\mathtt{s}$ if $\Gamma_2\vdash\mathtt{s}{:}\sigma(\tau_1)$, then $\Gamma_2\vdash\mathtt{s}{:}\tau_2$.

Then each composable term has a $<\#$-principal typing, where $<\#$ is the relation defined by rule (entail).

Proof. Let $\mathtt{s}$ be a composable term; by hypothesis $\mathtt{s}$ has a principal typing $t = \Gamma_1\vdash\tau_1$ s.t. properties 1 and 2 above hold.

We prove that t is also $<\#$-principal for $\mathtt{s}$. We already know by Theorem 2.4 that $<\#$ is sound, therefore for all t' $t<\#t' \Rightarrow$ $terms(t) \subseteq terms(t')$, hence $\mathtt{s} \in terms(t')$. For proving the opposite implication, let $t' = \Gamma_2\vdash\tau_2$ be s.t. $\mathtt{s} \in terms(t')$. Then we have the following implications:

$\mathtt{s} \in terms(t') \Rightarrow$
(t is principal for $\mathtt{s}$)
$t \leq t' \Rightarrow$
(by properties 1 and 2 and completeness
of the entailment and subtyping relations)
$\exists\,\sigma$ s.t. $\Gamma_2<\#\sigma(\Gamma_1)$ and $\Gamma_2 \vdash \sigma(\tau_1)<\#\tau_2 \Rightarrow$
(by rule (*entail*))
$t<\#t'$.

□

In the hypothesis of Theorem 2.5, we have the following two corollaries:

(1) Linking $\vdash^{<\#}$, with $<\#$ defined by rule (*entail*), induces a sound compositional compilation for all typings (by Theorems 2.3 and 2.4);
(2) Compositional compilation $\vdash_C^{<\#}$ with $<\#$ defined by rule (*entail*) is sound and complete (by Theorems 2.1 and 2.5).

It is interesting to notice that in general properties 1. and 2. of Theorem 2.5 are not expected to hold for all pairs of typings t_1, t_2 s.t. $t_1 \leq t_2$; this is due to the fact that rule (*entail*) is sound but not complete w.r.t. $\leq$. However, recalling Remark 2.1 in Sect.2.3, this fact does not prevent the typable terms of the type system to have $<\#$-principal typings for a sound relation $<\#$ which does not coincide with $\leq$.

3. Examples of instantiations of the framework

In this section we show how our framework can be instantiated with quite different type systems.

In order to prove the generality of our approach, we consider instantiations of the framework both for functional and object-oriented languages, choosing the λ-calculus and Featherweight Java as representative of the two paradigms, respectively. We also show how the framework can be used to guarantee compositionality in the presence of binary code transformation.

3.1. *Simply typed λ-calculus*

As a first example of instantiation of the framework we consider the simply typed λ-calculus (STLC for short, for the details see the Appendix). In this very simple case a source fragment is a λ-term, and τ denotes just a type and not binary code, since here we are interested in type inference, rather than in the whole compilation process.

It is well-known[9] that STLC has principal typings. For instance, let us consider the following source fragment:
$[f \triangleright \lambda x.(g\ (g\ x)), g \triangleright \lambda x.(h\ (h\ c)), h \triangleright \lambda x.c]$
Then we can derive the following principal typings for the three terms named f, g and h, respectively:

- $t_f = g : \alpha \to \alpha \vdash \alpha \to \alpha$ for $\lambda x.(g\ (g\ x))$
- $t_g = h : \kappa \to \kappa \vdash \alpha \to \kappa$ for $\lambda x.(h\ (h\ c))$

- $t_h = \emptyset \vdash \alpha \to \kappa$ for $\lambda x.c$

If we consider the linkset formed by the three principal typings above

$$\mathtt{L} = [f \triangleright t_f, g \triangleright t_g, h \triangleright t_h]$$

then we can derive the following linking judgment:

$$\vdash_C \mathtt{L}{:}[f \triangleright \kappa \to \kappa, g \triangleright \kappa \to \kappa, h \triangleright \kappa \to \kappa]$$

It is not difficult to prove that the typings t_f, t_g and t_h are all stronger than $f : \kappa \to \kappa, g : \kappa \to \kappa, h : \kappa \to \kappa \vdash \kappa \to \kappa$. Clearly, if we compile the three functions globally we get the same result.

We show on this example how the linking judgment is defined in terms of the entailment $<\#$ defined by rule (*entail*) (see the Appendix for the straight definitions of the entailment between environments and of subtyping).

By definition the linking judgment above is valid iff $t_f <\# t$, $t_g <\# t$ and $t_h <\# t$ are valid, with $t = f : \kappa \to \kappa, g : \kappa \to \kappa, h : \kappa \to \kappa \vdash \kappa \to \kappa$. The validity of the three relations can be obtained by applying the substitution σ mapping α to κ; for instance, if $\Gamma = f : \kappa \to \kappa, g : \kappa \to \kappa, h : \kappa \to \kappa$ and $\tau = \kappa \to \kappa$, $\Gamma_f = g : \alpha \to \alpha$ and $\tau_f = \alpha \to \alpha$, then $\Gamma <\# \sigma(\Gamma_f)$ and $\Gamma \vdash \sigma(\tau_f) <\# \tau$.

3.2. *Compositional compilation in Java*

Solutions to the problem of compositional compilation of Java has been already proposed.[6,7] Here we show how those results can be re-casted in the general framework defined in this paper.

We first show that type systems for Java-like languages based on conventional Java separate compilation fail to be compositional. For doing that we consider Featherweight Java (FJ for short),[10] a standard calculus for Java-like languages.

In the FJ examples below, `s` will denote a single FJ class declaration. Furthermore, types as defined in FJ are extended in order to include a notion of bytecode, since type annotations, needed by the verifier, make Java compositional compilation a hard task.

For instance, let us consider the following class declaration:

```
class A extends Object{
 E m(B x){return x.f1.f2;}}
```

The fact that class `A` can be successfully compiled in an environment containing class `B` with field `f1` of type `C`, and class `C` with field `f2` of type `E` can be formalized by the validity of the typing judgment $\Gamma \vdash \mathtt{s}_A : \tau$ where

- Γ=`B:class B extends Object{C f1;},`
  `C:class C extends Object{E f2;},`
  `E:class E extends Object{}`
- $\mathtt{s}_A$ is the declaration of `A` as above;
- τ=`class A extends Object{`
  `E m(B x){return x[B.f1 C][C.f2 E]}}`.

The type of a class contains, as expected, all usual type information (the name of the class and of its direct superclass, the name and type of declared fields, and the name and signature of declared methods). Furthermore, the type contains the generated bytecode for each declared method: for instance the bytecode of the method of `A` contains annotations which reflect the classes where fields were found and their types.

However, the typing $\Gamma \vdash \tau$ is not the only valid typing for $\mathtt{s}_A$. For instance, class `A` can be successfully compiled also in an environment containing a class `B` with a field `f1` of type `D`, and a class `D` with a field `f2` of type `F`, for some `F` subclass of `E`. Formally, $\Gamma' \vdash \tau'$ is another valid typing for $\mathtt{s}_A$, where

- Γ'=`B:class B extends Object{D f1;},`
  `D:class D extends Object{F f2;},`
  `F:class F extends E{},`
  `E:class E extends Object{}`
- τ'=`class A extends Object{`
  `E m(B x){return x[B.f1 D][D.f2 F]}}`.

Now one can prove that in FJ neither of the two typings above is principal for $\mathtt{s}_A$; indeed, the two typings are not comparable (neither is stronger than the other), and there is no typing of $\mathtt{s}_A$ which is stronger than both $\Gamma \vdash \tau$ and $\Gamma' \vdash \tau'$. The former claim can be easily proved by showing a class declaration for which $\Gamma \vdash \tau$ is a valid typing but not $\Gamma' \vdash \tau'$, and conversely; the latter claim relies on the (provable) intuition that in FJ a typing $\Gamma \vdash \tau$ is stronger than $\Gamma' \vdash \tau'$ iff $\Gamma' <\# \Gamma$, where $<\#$ basically corresponds to width and depth record subtyping. Then, it is not difficult to prove that there is no way to weaken Γ or Γ' (hence to find an environment Γ'' s.t. $\Gamma, \Gamma' <\# \Gamma''$) without losing typability of $\mathtt{s}_A$ in FJ.

The solution[7] to achieve principality consists in adding type constraints and type variables (for avoiding confusion we call FJP the calculus obtained from this extension). For instance, in FJP the class declaration $\mathtt{s}_A$ as above has the principal typing $\Gamma_A \vdash \tau_A$ where

- Γ_A=ϕ(B,f1,α),ϕ(α,f2,β),$\beta \leq$ E
- τ_A=class A extends Object{
  E m(B x){return x[B.f1 α][α.f2 β]}}.

A constraint of the shape ϕ(T1,f,T2) requires[d] type T1 to declare or inherit a field f of type T2, while α and β are type variables. Note that α and β occur both in the type environment and in the generated bytecode. Indeed, compilation generates *polymorphic bytecode*, a generalized form of bytecode which needs to be instantiated into conventional bytecode before being correctly executed by the Java Virtual Machine (JVM).

Finally, the constraint T1 $\leq$ T2 requires T1 to be a subtype (in the Java sense) of T2. It is important not to confuse Java subtyping with the subtyping relation <# introduced in Sect.2.4; indeed, in this example of instantiation of the framework the two relations do not coincide (see below).

Now we can show how a sequence of class declarations can be compiled compositionally. Let $\mathtt{s}_B$ and $\mathtt{s}_E$ denote the following class declarations, respectively:

```
class B extends Object{
 E f1; B m1(){return this;}}
class E extends B{
 E f2; B m2(A x){return x.m(this.f1);}}
```

If $\Gamma_B, \tau_B, \Gamma_E, \tau_E$ are defined as follows:

- $\Gamma_B = \exists$E
- τ_B=class B extends Object{
  E f1; B m1(){return this;}}
- Γ_E=B $\not<$ E, ϕ(E,f1,α), μ(A,m,α,(β,α')),$\beta \leq$ B
- τ_E=class E extends B{
  E f2;
  B m2(A x){
  return x [A.m(α') β] (this [E.f1 α]);}}

[d]Analogous constraints are introduced for dealing with the other constructs, namely, method invocation, object creation, and type cast.

Then $\Gamma_B \vdash \tau_B$ and $\Gamma_E \vdash \tau_E$ are principal for $\mathtt{s}_B$ and $\mathtt{s}_E$, respectively.

Linking on the linkset

$$\mathtt{L} = [A \triangleright \Gamma_A \vdash \tau_A, B \triangleright \Gamma_B \vdash \tau_B, E \triangleright \Gamma_E \vdash \tau_E]$$

is modeled by the following valid judgment:

$$\vdash_C \mathtt{L}{:}A \triangleright \tau'_A, B \triangleright \tau'_B, E \triangleright \tau'_E$$

where τ'_A and τ'_E are obtained by applying the substitution $\mathtt{[E/}\alpha\mathtt{,\ B/}\alpha'\mathtt{,\ E/}\beta\mathtt{]}$ to τ_A and τ_E, respectively, while τ'_B is equal to τ_B. Then by rule (*comp*) we derive the judgment $\vdash_C [A \triangleright \mathtt{s}_A, B \triangleright \mathtt{s}_B, E \triangleright \mathtt{s}_E]{:}A \triangleright \tau'_A, B \triangleright \tau'_B, E \triangleright \tau'_E$, thus obtaining the same set of binaries generated by a global compilation.

Finally, we would like to point out that it can be proved that every term in FJP is typable, but of course not all terms are composable; typings of non composable terms are always of the form $\Gamma \vdash \tau$, where Γ is unsatisfiable, that is, there is no statically correct program corresponding to Γ, like happens, for instance, for $\Gamma = \mathtt{c}_1 \leq \mathtt{c}_2, \mathtt{c}_2 \leq \mathtt{c}_1$, with $\mathtt{c}_1$ and $\mathtt{c}_2$ distinct class names.

3.3. *Compositional compilation and binary code transformation*

Binary code transformation/optimization is often in contrast with compositional compilation, since many times it requires some form of global analysis. However, by introducing some intermediate form of more abstract binary code one can recover compositionality. As suggested by rule (*entail*) given in Sect.2.4, at linking-time the code generated for each single source fragment is expected to be modified in two different ways: first some type variables might need to be instantiated; second, the type obtained by substitution might need to be replaced with a supertype.

In the example in Sect.3.2 substitutions apply both to type information and to code, and a non trivial subtyping relation is defined since, for instance, $\Gamma \vdash \ll \mathtt{c}, \mathtt{t} \gg \mathtt{e}^\mathtt{b} < \#\mathtt{e}^\mathtt{b}$ is valid whenever $\Gamma < \#\mathtt{t} \leq \mathtt{c}$ is valid. The fact that $\ll \mathtt{c}, \mathtt{t} \gg \mathtt{e}^\mathtt{b}$ is a subtype of $\mathtt{e}^\mathtt{b}$ whenever $\Gamma < \#\mathtt{t} \leq \mathtt{c}$, corresponds to the intuition that there is a loss of information when $\ll \mathtt{c}, \mathtt{t} \gg \mathtt{e}^\mathtt{b}$ is transformed into $\mathtt{e}^\mathtt{b}$, even though the transformation preserves the semantics. More formally, we expect subtyping to be sound according to Def. 2.8.

In a sense, specializing $\ll \mathtt{c}, \mathtt{t} \gg \mathtt{e}^\mathtt{b}$ into $\mathtt{e}^\mathtt{b}$ is already an example of code optimization. However here we consider a more meaningful example of code transformation, namely, auto-boxing and -unboxing of integer values and objects, respectively; we refer to[11] for other interesting just-in-time type

specializations in the context of Java-like languages. The following example of correct Java 1.5 source code illustrates the use of auto-boxing and -unboxing:

```
class C{
 void m1(int i){}
 void m2(Integer i){}
 void m3(){m2(1);m1(new Integer(1));}}
```

While processing the invocation of `m2` the compiler automatically coerces the expression `1` to have type `Integer` by wrapping `1` with `new Integer()`. Conversely, while processing the invocation of `m1` the compiler automatically coerces the expression `new Integer(1)` to have type `int` by invoking method `intValue()` of class `Integer`.

In order to model auto-boxing and -unboxing in FJP we need to add integer literals and the primitive type `int` to the source syntax, to assume that besides `Object`, there is also the predefined `Integer` class with its corresponding field and method, and, more importantly, we need to add a new form of binary method call where type annotation includes also the types of the arguments:

$$\begin{array}{c} \mathsf{e}^{\mathsf{b}} ::= \ldots \mid \text{(all FJP binary expressions)} \\ \mathsf{e}^{\mathsf{b}}_0[\mathtt{t}(\bar{\mathtt{t}}).\mathsf{m}(\bar{\mathtt{t}}')\mathtt{t}'](\mathsf{e}^{\mathsf{b}}_1 \ldots \mathsf{e}^{\mathsf{b}}_n) \end{array}$$

where $\bar{\mathtt{t}}$ are the types of the arguments, whereas $\bar{\mathtt{t}}'$ are the expected types of the parameters.

Then, the following rule has to be added:

$$\frac{\begin{array}{ll} \Gamma\vdash \mathsf{e}^{\mathsf{s}}_0 : (\mathtt{t}_0,\mathsf{e}^{\mathsf{b}}_0) & \\ \Gamma\vdash \mathsf{e}^{\mathsf{s}}_i : (\mathtt{t}_i,\mathsf{e}^{\mathsf{b}}_i) & \forall i \in 1..n \\ \bar{\mathtt{t}} = \mathtt{t}_1,\ldots,\mathtt{t}_n & \Gamma{<}\#\mu(\mathtt{t}_0,\mathsf{m},(\bar{\mathtt{t}}),(\mathtt{t},\bar{\mathtt{t}}')) \end{array}}{\Gamma\vdash \mathsf{e}^{\mathsf{s}}_0.\mathsf{m}(\mathsf{e}^{\mathsf{s}}_1,\ldots,\mathsf{e}^{\mathsf{s}}_n) : (\mathtt{t},\mathsf{e}^{\mathsf{b}}_0[\mathtt{t}_0(\bar{\mathtt{t}}).\mathsf{m}(\bar{\mathtt{t}}')\mathtt{t}](\mathsf{e}^{\mathsf{b}}_1,\ldots,\mathsf{e}^{\mathsf{b}}_n))}$$

For what concerns environment entailment, we need to specify that types `int` and `Integer` are interchangeable:

$$\frac{}{\Gamma{<}\#(\mathtt{int} \leq \mathtt{Integer})} \qquad \frac{}{\Gamma{<}\#(\mathtt{Integer} \leq \mathtt{int})}$$

Finally, we add the new subtyping rule which allows specialization of the generalized binary method invocation into standard bytecode, where u ranges over ground types (either a class or `int`):

$$\frac{\Gamma{<}\#u_i \leq u'_i, i = 1..n}{\Gamma\vdash \mathsf{e}^{\mathsf{b}}_0[u_0(\bar{u}).\mathsf{m}(\bar{u}')u](\bar{\mathsf{e}}^{\mathsf{b}}){<}\#\mathsf{e}^{\mathsf{b}}_0[u_0.\mathsf{m}(\bar{u}')u](\bar{\mathsf{e}}^{\mathsf{b}}_{\mathsf{T}})}$$

where

$$\bar{u} = u_1, \ldots, u_n \qquad \bar{u}' = u'_1, \ldots, u'_n$$
$$\bar{\mathsf{e}}^{\mathsf{b}} = \mathsf{e}^{\mathsf{b}}_1, \ldots, \mathsf{e}^{\mathsf{b}}_n \qquad \bar{\mathsf{e}}^{\mathsf{b}}_{\mathsf{T}} = \mathsf{e}^{\mathsf{b}}_{n+1}, \ldots, \mathsf{e}^{\mathsf{b}}_{2n}$$
for $i \in 1..n$

$$\mathsf{e}^{\mathsf{b}}_{n+i} = \begin{cases} \texttt{new Integer}(\mathsf{e}^{\mathsf{b}}_i) & \text{if } u_i = \texttt{int} \text{ and } u_i \neq u'_i \\ \mathsf{e}^{\mathsf{b}}_i\texttt{.intValue()} & \text{if } u_i = \texttt{Integer} \\ & \text{and } u'_i = \texttt{int} \\ \mathsf{e}^{\mathsf{b}}_i & \text{otherwise} \end{cases}$$

As a last remark, we would to note that this extension can be nicely integrated with the existing type system, thanks to the adopted modular approach.

4. Conclusion

The contributions of the paper can summarized as follows:

(1) A system-independent formal definition of compositional compilation (separate compilation and linking) and its expected properties. This work has been inspired by the seminal paper,[5] where, however, definitions were given for a fixed language (a simple lambda calculus) and issues of soundness and completeness were not considered.
(2) A sufficient condition for soundness and completeness of compositional compilation, that is, the existence of an entailment relation on typings s.t. for each composable fragment there is a typing which entails all and only all the typings of the fragment. This condition is weaker than the principal typings[9] property, and coincides with it for relations which are sound w.r.t. typability of all terms.
(3) A modular way of defining an entailment relation between typings on top of an entailment relation between type environments and a subtyping relation which satisfy in turn appropriate soundness and completeness requirements.
(4) Instantiations of the framework on three rather different type systems: simply typed lambda calculus, Featherweight Java and an extension of Featherweight Java with a boxing/unboxing mechanism.

Concerning the second point, we believe the result is significant since it shows that an independent characterization, designed starting from the intuition on what a linking procedure should guarantee, is then discovered to be equivalent to the principal typings property, thus confirming that this property is the right one when we want both sound and complete compositional compilation and soundness of the entailment relation. In this case, we

have the additional benefit that linking is safe for all typings, and not only for those principal. However, it is interesting to note that, for only achieving sound and complete compositional compilation, one could in principle rely on the weaker property of $<\#$-principal typings for some $<\#$ entailment relation: in this case, if the type information carried by a fragment is $<\#$-principal, then linking succeeds if and only if global recompilation does; however, linking of arbitrary typings might not be safe.

Concerning the last point, an important result is that we have shown how the notion of type specialization, which is widely used in functional languages, can be nicely extended to include binary code specialization as well. More in general, our work demonstrates that many classical notions from type theory, where compilation is usually simplified to the hardest part, that is, type inference, can be reformulated in those contexts where binary code generation becomes an issue which cannot be neglected to guarantee compositionality. What happens is that, to achieve compositional compilation, one needs to define more abstract forms of binary code, as happens for types in compositional analysis.

We have provided here a simple preliminary example based on this idea, which we believe can be applied to a variety of code transformations (as optimization) in different languages.

Appendix A. Main definitions for STLC and FJP

$$\mathtt{s} ::= c \mid x \mid (\mathtt{s}_1\mathtt{s}_2) \mid (\lambda x.\mathtt{s})$$
$$\tau ::= \kappa \mid \alpha \mid \tau_1 \to \tau_2$$

Fig. A1. Syntax of STLC

$$con\frac{}{\Gamma \vdash c : \kappa} \qquad var\frac{}{\Gamma \vdash x : \tau}\ x : \tau \in \Gamma$$

$$app\frac{\Gamma \vdash \mathtt{s}_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash \mathtt{s}_2 : \tau_1}{\Gamma \vdash (\mathtt{s}_1 \mathtt{s}_2) : \tau_2}$$

$$abs\frac{\Gamma, x : \tau_1 \vdash \mathtt{s} : \tau_2}{\Gamma \vdash (\lambda x.\mathtt{s}) : \tau_1 \to \tau_2}$$

Fig. A2. Typing rules for STLC

$$\frac{}{\Gamma < \#\Gamma'}\ \Gamma' \subseteq \Gamma \qquad \frac{}{\Gamma \vdash \tau < \#\tau}$$

Fig. A3. Environment entailment and subtyping for STLC

$$\begin{aligned}
\mathsf{s} &::= \mathtt{class}\ \mathsf{c}\ \mathtt{extends}\ \mathsf{c}'\ \{\ \mathsf{fds}\ \mathsf{mds}^{\mathsf{s}}\ \}\ \ (\mathsf{c} \neq \mathtt{Object}) \\
\mathsf{fds} &::= \mathsf{c}_1\ \mathsf{f}_1\,;\ldots \mathsf{c}_n\ \mathsf{f}_n\,; \\
\mathsf{mds}^{\mathsf{s}} &::= \mathsf{md}^{\mathsf{s}}_1 \ldots \mathsf{md}^{\mathsf{s}}_n \\
\mathsf{md}^{\mathsf{s}} &::= \mathsf{mh}\ \{\mathtt{return}\ \mathsf{e}^{\mathsf{s}}\,;\} \\
\mathsf{mh} &::= \mathsf{c}_0\ \mathsf{m}(\mathsf{c}_1\ \mathsf{x}_1 \ldots \mathsf{c}_n\ \mathsf{x}_n) \\
\mathsf{e}^{\mathsf{s}} &::= \mathsf{x} \mid \mathsf{e}^{\mathsf{s}}.\mathsf{f} \mid \mathsf{e}^{\mathsf{s}}_0.\mathsf{m}(\mathsf{e}^{\mathsf{s}}_1 \ldots \mathsf{e}^{\mathsf{s}}_n) \mid \mathtt{new}\ \mathsf{c}(\mathsf{e}^{\mathsf{s}}_1 \ldots \mathsf{e}^{\mathsf{s}}_n) \mid (\mathsf{c})\mathsf{e}^{\mathsf{s}}
\end{aligned}$$

where field, method and parameter names
are distinct in fds, $\mathsf{mds}^{\mathsf{s}}$ and mh

Fig. A4. Syntax of FJP source language

References

1. W. F. Tichy, *ACM Transactions on Programming Languages and Systems* **8**, 273 (1986).
2. R. W. Schwanke and G. E. Kaiser, *ACM Transactions on Programming Languages and Systems* **10**, 627 (1988).
3. Z. Shao and A. Appel, Smartest recompilation, in *ACM Symp. on Principles of Programming Languages 1993*, (ACM Press, 1993).
4. R. Adams, W. Tichy and A. Weinert, *ACM Transactions on Software Engineering and Methodology* **3**, 3 (1994).
5. L. Cardelli, Program fragments, linking, and modularization, in *ACM Symp. on Principles of Programming Languages 1997*, (ACM Press, 1997).
6. D. Ancona and E. Zucca, Principal typings for Java-like languages, in *ACM*

$$
\begin{array}{rl}
\tau & ::= \texttt{class c extends c}' \ \{ \ \mathsf{fds}\ \mathsf{mds}^{\mathsf{b}} \ \} \ \ (\texttt{c} \neq \texttt{Object}) \\
\mathsf{mds}^{\mathsf{b}} & ::= \mathsf{md}_1^{\mathsf{b}} \ldots \mathsf{md}_n^{\mathsf{b}} \\
\mathsf{md}^{\mathsf{b}} & ::= \mathsf{mh}\ \{\texttt{return}\ \mathsf{e}^{\mathsf{b}}\texttt{;}\} \\
\mathsf{e}^{\mathsf{b}} & ::= \texttt{x} \ \mid \ \mathsf{e}^{\mathsf{b}}[\texttt{t.f t}'] \mid \ \mathsf{e}_0^{\mathsf{b}}[\texttt{t.m}(\bar{\texttt{t}})\texttt{t}'](\mathsf{e}_1^{\mathsf{b}} \ldots \mathsf{e}_n^{\mathsf{b}}) \mid \\
& \qquad \texttt{new}\ [\texttt{c}\ \bar{\texttt{t}}](\mathsf{e}_1^{\mathsf{b}} \ldots \mathsf{e}_n^{\mathsf{b}}) \mid \ (\texttt{c})\mathsf{e}^{\mathsf{b}} \ \mid \ \ll \texttt{c}, \texttt{t} \gg \mathsf{e}^{\mathsf{b}} \\
\texttt{t} & ::= \texttt{c} \ \mid \ \alpha \\
\bar{\texttt{t}} & ::= \texttt{t}_1 \ldots \texttt{t}_n
\end{array}
$$

where fds and mh are defined in Fig.A4 and
method names in $\mathsf{mds}^{\mathsf{b}}$ are distinct

Fig. A5. Syntax of FJP polymorphic bytecode

$$
\begin{array}{rl}
\Gamma & ::= \emptyset \mid \gamma, \Gamma \\
\gamma & ::= \texttt{t} \leq \texttt{t}' \ \mid \exists \texttt{c} \ \mid \phi(\texttt{t}, \texttt{f}, \texttt{t}') \mid \mu(\texttt{t}, \texttt{m}, \bar{\texttt{t}}, (\texttt{t}', \bar{\texttt{t}}')) \mid \\
& \quad \kappa(\texttt{c}, \bar{\texttt{t}}, \bar{\texttt{t}}') \mid \texttt{c} \sim \texttt{t} \mid \texttt{c} \not< \texttt{c}' \mid \texttt{x} : (\texttt{c}, \mathsf{e}^{\mathsf{b}}) \mid \\
& \quad \texttt{c} : \texttt{class c extends c}' \ \{ \ \mathsf{fds}\ \mathsf{mds}^{\mathsf{b}} \ \} \ \ (\texttt{c} \neq \texttt{Object})
\end{array}
$$

where $\exists\, \texttt{c}$ is just a convenient shortcut for $\texttt{c} \leq \texttt{c}$

Fig. A6. Type environments for FJP

Symp. on Principles of Programming Languages 2004, (ACM Press, January 2004).

7. D. Ancona, F. Damiani, S. Drossopoulou and E. Zucca, Polymorphic bytecode: Compositional compilation for Java-like languages, in *ACM Symp. on Principles of Programming Languages 2005*, (ACM Press, January 2005).
8. T. Jim, What are principal typings and what are they good for?, in *ACM Symp. on Principles of Programming Languages 1996*, (ACM Press, 1996).
9. J. B. Wells, The essence of principal typings, in *International Colloquium on Automata, Languages and Programming 2002*, Lecture Notes in Computer Science(2380) (Springer, 2002).
10. A. Igarashi, B. C. Pierce and P. Wadler, *ACM Transactions on Programming Languages and Systems* **23**, 396 (2001).
11. A. Kennedy and D. Syme, Design and implementation of generics for the .net common language runtime, in *PLDI'01 - ACM Conf. on Programming Language Design and Implementation*, (ACM Press, New York, NY, USA, 2001).

$$\frac{\Gamma \vdash \mathsf{mds}_1^{\mathsf{b}} <: \# \mathsf{mds}_2^{\mathsf{b}}}{\Gamma \vdash \texttt{class c extends c}' \texttt{ \{ fds } \mathsf{mds}_1^{\mathsf{b}} \texttt{ \}} <: \# \texttt{class c extends c}' \texttt{ \{ fds } \mathsf{mds}_2^{\mathsf{b}} \texttt{ \}}}$$

$$\frac{\Gamma \vdash \mathsf{md}_i^{\mathsf{b}} <: \# \mathsf{md}_{i+n}^{\mathsf{b}} \ \forall i \in 1..n}{\Gamma \vdash \mathsf{md}_1^{\mathsf{b}} \ldots \mathsf{md}_n^{\mathsf{b}} <: \# \mathsf{md}_{1+n}^{\mathsf{b}} \ldots \mathsf{md}_{2n}^{\mathsf{b}}}$$

$$\frac{\Gamma \vdash \mathsf{e}^{\mathsf{b}} <: \# \mathsf{e}^{\mathsf{b}'}}{\Gamma \vdash \mathsf{mh} \ \{\texttt{return } \mathsf{e}^{\mathsf{b}}\texttt{;}\} <: \# \mathsf{mh} \ \{\texttt{return } \mathsf{e}^{\mathsf{b}'}\texttt{;}\}}$$

$$\frac{\Gamma \vdash \mathsf{e}^{\mathsf{b}} <: \# \mathsf{e}^{\mathsf{b}'}}{\Gamma \vdash (\mathtt{t}, \mathsf{e}^{\mathsf{b}}) <: \# (\mathtt{t}, \mathsf{e}^{\mathsf{b}'})} \qquad \frac{}{\Gamma \vdash \mathsf{e}^{\mathsf{b}} <: \# \mathsf{e}^{\mathsf{b}}}$$

$$\frac{\Gamma <: \# \mathtt{c} \leq \mathtt{t}}{\Gamma \vdash \ll \mathtt{c}, \mathtt{t} \gg \mathsf{e}^{\mathsf{b}} <: \# (\mathtt{c}) \mathsf{e}^{\mathsf{b}}} \qquad \frac{\Gamma <: \# \mathtt{t} \leq \mathtt{c}}{\Gamma \vdash \ll \mathtt{c}, \mathtt{t} \gg \mathsf{e}^{\mathsf{b}} <: \# \mathsf{e}^{\mathsf{b}}}$$

Fig. A7. Subtyping for FJP

$$\subseteq\frac{}{\Gamma<\#\Gamma'}\ \Gamma'\subseteq\Gamma \qquad \leq\text{-}refl\frac{}{\Gamma<\#\mathsf{c}\leq\mathsf{c}}\ c:\tau\in\Gamma \qquad \leq\text{-}Obj\frac{}{\Gamma<\#\texttt{Object}\leq\texttt{Object}}$$

$$\leq\text{-}trans\frac{\Gamma<\#\mathsf{c}_2\leq\mathsf{c}_3}{\Gamma<\#\mathsf{c}_1\leq\mathsf{c}_3}\ \mathsf{c}_1:\texttt{class}\ \mathsf{c}_1\ \texttt{extends}\ \mathsf{c}_2\ \{\ \mathsf{fds}\ \mathsf{mds}^{\mathsf{b}}\ \}\in\Gamma$$

$$\phi\text{-}1\frac{}{\Gamma<\#\phi(\mathsf{c},\mathsf{f},\mathsf{c}'')}\ \begin{array}{l}\mathsf{c}:\tau\in\Gamma\\ \mathsf{c}''\ \mathsf{f};\in\tau\end{array} \qquad \phi\text{-}2\frac{\Gamma<\#\phi(\mathsf{c}',\mathsf{f},\mathsf{c}'')}{\Gamma<\#\phi(\mathsf{c},\mathsf{f},\mathsf{c}'')}\ \begin{array}{l}\mathsf{c}:\texttt{class}\ \mathsf{c}\ \texttt{extends}\ \mathsf{c}'\ \{\ \mathsf{fds}\ \mathsf{mds}^{\mathsf{b}}\ \}\in\Gamma\\ \mathsf{f}\notin\mathsf{fds}\end{array}$$

$$\mu\text{-}1\frac{\Gamma<\#\mathsf{c}_i\leq\mathsf{c}_i''\ \forall\, i\in 1..n}{\Gamma<\#\mu(\mathsf{c},\mathsf{m},\mathsf{c}_1\ldots\mathsf{c}_n,(\mathsf{c}'',\mathsf{c}_1''\ldots\mathsf{c}_n''))}\ \begin{array}{l}\mathsf{c}:\tau\in\Gamma\\ \mathsf{c}''\ \mathsf{m}(\mathsf{c}_1''\ldots\mathsf{c}_n'')\in\tau\end{array}$$

$$\mu\text{-}2\frac{\Gamma<\#\mu(\mathsf{c}',\mathsf{m},\bar{\mathsf{c}},(\mathsf{c}'',\bar{\mathsf{c}}''))}{\Gamma<\#\mu(\mathsf{c},\mathsf{m},\bar{\mathsf{c}},(\mathsf{c}'',\bar{\mathsf{c}}''))}\ \begin{array}{l}\mathsf{c}:\texttt{class}\ \mathsf{c}\ \texttt{extends}\ \mathsf{c}'\ \{\ \mathsf{fds}\ \mathsf{mds}^{\mathsf{b}}\ \}\in\Gamma\\ \mathsf{m}\notin\mathsf{mds}^{\mathsf{b}}\end{array}$$

$$\kappa\text{-}1\frac{}{\Gamma<\#\kappa(\texttt{Object},\epsilon,\epsilon)}$$

$$\kappa\text{-}2\frac{\begin{array}{l}\Gamma<\#\kappa(\mathsf{c}',\mathsf{c}_1'\ldots\mathsf{c}_k',\mathsf{c}_1\ldots\mathsf{c}_k)\\ \Gamma<\#\mathsf{c}_i'\leq\mathsf{c}_i\ \forall\, i\in k+1..n\end{array}}{\Gamma<\#\kappa(\mathsf{c},\mathsf{c}_1'\ldots\mathsf{c}_n',\mathsf{c}_1\ldots\mathsf{c}_n)}\ \mathsf{c}:\texttt{class}\ \mathsf{c}\ \texttt{extends}\ \mathsf{c}'\ \{\ \mathsf{c}_{k+1}\ \mathsf{f}_{k+1};\ldots\mathsf{c}_n\ \mathsf{f}_n;\ \mathsf{mds}^{\mathsf{b}}\ \}\in\Gamma$$

$$\sim\text{-}1\frac{\Gamma<\#\mathsf{c}\leq\mathsf{c}'}{\Gamma<\#\mathsf{c}\sim\mathsf{c}'} \qquad \sim\text{-}2\frac{\Gamma<\#\mathsf{c}\sim\mathsf{c}'}{\Gamma<\#\mathsf{c}'\sim\mathsf{c}}$$

$$\not\sim\text{-}1\frac{}{\Gamma<\#\mathsf{c}\not\sim\mathsf{c}'}\ \mathsf{c}:\tau\notin\Gamma \qquad \not\sim\text{-}2\frac{\Gamma<\#\mathsf{c}'\not\sim\mathsf{c}''}{\Gamma<\#\mathsf{c}\not\sim\mathsf{c}''}\ \begin{array}{l}\mathsf{c}:\texttt{class}\ \mathsf{c}\ \texttt{extends}\ \mathsf{c}'\ \{\ \mathsf{fds}\ \mathsf{mds}^{\mathsf{b}}\ \}\in\Gamma\\ \mathsf{c}'\neq\mathsf{c}''\end{array}$$

Fig. A8. Environment entailment for FJP

$$\mathit{class}\frac{\Gamma\vdash \mathsf{fds}{:}\mathsf{fds} \qquad \Gamma, \mathtt{this}{:}(\mathsf{c}, \mathtt{this})\vdash \mathsf{mds}^{\mathsf{s}} : \mathsf{mds}^{\mathsf{b}} \qquad \Gamma{<}\#\exists\, \mathsf{c}' \qquad \Gamma{<}\#\mathsf{c}' \not< \mathsf{c}}{\Gamma\vdash \mathtt{class}\ \mathsf{c}\ \mathtt{extends}\ \mathsf{c}'\ \{\mathsf{fds}\ \mathsf{mds}^{\mathsf{s}}\}{:}\mathtt{class}\ \mathsf{c}\ \mathtt{extends}\ \mathsf{c}'\ \{\mathsf{fds}\ \mathsf{mds}^{\mathsf{b}}\}}$$

$$\mathit{fields}\frac{\Gamma{<}\#\exists\, \mathsf{c}_i\ i \in 1..n}{\Gamma\vdash(\mathsf{c}_1\ \mathsf{f}_1\,;\, \ldots\, \mathsf{c}_n\ \mathsf{f}_n\,;){:}(\mathsf{c}_1\ \mathsf{f}_1\,;\, \ldots\, \mathsf{c}_n\ \mathsf{f}_n\,;)}$$

$$\mathit{methods}\frac{\Gamma\vdash \mathsf{md}_i^{\mathsf{s}} : \mathsf{md}_i^{\mathsf{b}} \quad \forall i \in 1..n}{\Gamma\vdash \mathsf{md}_1^{\mathsf{s}}\ \ldots\ \mathsf{md}_n^{\mathsf{s}} : \mathsf{md}_1^{\mathsf{b}} \ldots \mathsf{md}_n^{\mathsf{b}}}\ n \neq 1$$

$$\mathit{method}\frac{\Gamma, \mathsf{x}_1{:}(\mathsf{c}_1, \mathsf{x}_1) \ldots \mathsf{x}_n{:}(\mathsf{c}_n, \mathsf{x}_n)\vdash \mathsf{e}^{\mathsf{s}} : (\mathtt{t},\mathsf{e}^{\mathsf{b}}) \qquad \Gamma{<}\#\mathtt{t} \leq \mathsf{c}_0 \qquad \Gamma{<}\#\exists\, \mathsf{c}_i\ \forall i \in 0..n}{\Gamma\vdash \mathsf{c}_0\ \mathsf{m}(\mathsf{c}_1\ \mathsf{x}_1 \ldots \mathsf{c}_n\ \mathsf{x}_n)\ \{\mathtt{return}\ \mathsf{e}^{\mathsf{s}}\mathtt{;}\} : \mathsf{c}_0\ \mathsf{m}(\mathsf{c}_1\ \mathsf{x}_1 \ldots \mathsf{c}_n\ \mathsf{x}_n)\ \{\mathtt{return}\ \mathsf{e}^{\mathsf{b}}\mathtt{;}\}}$$

$$\mathit{parameter}\frac{}{\Gamma\vdash \mathsf{x} : (\mathsf{c},\mathsf{e}^{\mathsf{b}})}\ \mathsf{x} : (\mathsf{c}, \mathsf{e}^{\mathsf{b}}) \in \Gamma \qquad\qquad \mathit{field\ access}\frac{\Gamma\vdash \mathsf{e}^{\mathsf{s}} : (\mathtt{t},\mathsf{e}^{\mathsf{b}}) \qquad \Gamma{<}\#\phi(\mathtt{t}, \mathsf{f}, \mathtt{t}')}{\Gamma\vdash \mathsf{e}^{\mathsf{s}}.\mathsf{f} : (\mathtt{t}',\mathsf{e}^{\mathsf{b}}[\mathtt{t}.\mathsf{f}\ \mathtt{t}'])}$$

$$\mathit{meth\ call}\frac{\Gamma\vdash \mathsf{e}_0^{\mathsf{s}} : (\mathtt{t}_0,\mathsf{e}_0^{\mathsf{b}}) \qquad \Gamma\vdash \mathsf{e}_i^{\mathsf{s}} : (\mathtt{t}_i,\mathsf{e}_i^{\mathsf{b}}) \quad \forall i \in 1..n \qquad \Gamma{<}\#\mu(\mathtt{t}_0, \mathsf{m}, (\mathtt{t}_1, \ldots, \mathtt{t}_n), (\mathtt{t}, \bar{\mathtt{t}}))}{\Gamma\vdash \mathsf{e}_0^{\mathsf{s}}.\mathsf{m}(\mathsf{e}_1^{\mathsf{s}}, \ldots, \mathsf{e}_n^{\mathsf{s}}) : (\mathtt{t},\mathsf{e}_0^{\mathsf{b}}[\mathtt{t}_0.\mathsf{m}(\bar{\mathtt{t}})\mathtt{t}](\mathsf{e}_1^{\mathsf{b}}, \ldots, \mathsf{e}_n^{\mathsf{b}}))}$$

$$\mathit{new}\frac{\Gamma\vdash \mathsf{e}_i^{\mathsf{s}} : (\mathtt{t}_i,\mathsf{e}_i^{\mathsf{b}}) \quad \forall i \in 1..n \qquad \Gamma{<}\#\kappa(\mathsf{c}, \mathtt{t}_1 \ldots \mathtt{t}_n, \bar{\mathtt{t}})}{\Gamma\vdash \mathtt{new}\ \mathsf{c}(\mathsf{e}_1^{\mathsf{s}} \ldots \mathsf{e}_n^{\mathsf{s}}) : (\mathsf{c},\mathtt{new}\ [\mathsf{c}\ \bar{\mathtt{t}}](\mathsf{e}_1^{\mathsf{b}} \ldots \mathsf{e}_n^{\mathsf{b}}))}$$

$$\mathit{cast}\frac{\Gamma\vdash \mathsf{e}^{\mathsf{s}} : (\mathtt{t},\mathsf{e}^{\mathsf{b}}) \qquad \Gamma{<}\#\mathsf{c} \sim \mathtt{t}}{\Gamma\vdash (\mathsf{c})\mathsf{e}^{\mathsf{s}} : (\mathsf{c},{\ll}\mathsf{c}, \mathtt{t}{\gg}\ \mathsf{e}^{\mathsf{b}})} \qquad \mathit{sub}\frac{\Gamma\vdash \mathsf{e}^{\mathsf{s}} : (\mathsf{c},\mathsf{e}^{\mathsf{b}}) \qquad \Gamma\vdash(\mathsf{c}, \mathsf{e}^{\mathsf{b}}){<}\#(\mathsf{c}', \mathsf{e}^{\mathsf{b}'})}{\Gamma\vdash \mathsf{e}^{\mathsf{s}} : (\mathsf{c}',\mathsf{e}^{\mathsf{b}'})}$$

Fig. A9. Typing rules for FJP