Safety in Flexible Dynamic Linking

Alex Buckley and Sophia Drossopoulou
Department of Computing, Imperial College London
{abuckley,scd}@doc.ic.ac.uk

Abstract

Dynamic linking lets programs use the most recent versions of classes without re-compilation. In Java and .NET, bytecode specifies which classes should be dynamically linked. This information represents the compiler's knowledge of the compilation environment, but the execution environment might be different. For example, the execution environment on a mobile phone might provide fewer, simpler classes than on a desktop PC. As bytecode cannot adapt to its execution environment, component reuse is restricted and development costs are increased.

We suggest *flexible* dynamic linking that binds bytecode as late as possible to the classes available in an execution environment. Rather than specifying classes, bytecode contains type variables that are substituted by classes as late as execution. We present a non-deterministic model that treats substitution as a linking step, and interleaves linking with execution. We examine how linking builds a well-formed program and prove soundness for the laziest substitution strategy, representing the most general execution.

1 Introduction

Most language mechanisms for modular development - packages in Ada, modules in Modula-2, classes and namespaces in C++ - have no role at run-time. A compiler typically employs a static linker to emit a monolithic executable file, so the compilation environment automatically becomes the entire execution environment. Few (or no) dynamic checks are needed to resolve external dependencies.

In contrast, the basic unit of development in Java and C# - the class - maintains its discrete identity throughout compilation and execution. A Java Virtual Machine or Microsoft's .NET Common Language Runtime can start with the bytecode for just one class, then lazily load and link classes from the execution environment as necessary for continued execution. The exact execution environment is not generally known in advance, and may differ from the compilation environment. Dynamic checking, often in the guise of a verification stage, is required to ensure sound linking.

However, while execution is lazy, compilers are not. To enable a class to safely link against its environment *at run-time*, they embed type information about other classes into its bytecode *at compile-time*. Thus, they implicitly embed the "shape" of the compilation environment. For example, compiling `new A().f.g;` in the compilation environment:

```
class A { B f; }
class B { C g; }
```

results in this (high-level representation of) bytecode:

```
1  new <Class A>
...
5  loadfield <Field B f>
6  loadfield <Field C g>
```

Running the bytecode in an execution environment E_1 that is identical to the compilation environment will obviously succeed. But if we run the bytecode in the execution environment E_2:

```
class A { D f; }
class D { C g; }
```

then we can expect it to fail since it demands a field *of type* `B` in class `A`, whereas E_2 provides a field of type `D` in class `A`. This failure is unsatisfactory because the programmer's intention, expressed in the final bytecode instruction, was to access a field `g` of type `C`, and E_2 provides such a field.

We expect more and more scenarios where the execution environment differs from the compilation environment, and thus a choice arises as to which classes should be dynamically linked [AGW04]. Already, a common situation is that frameworks allow vendor-provided implementations of common interfaces, *e.g.* as in the JDBC API. The ability to dynamically choose and use the "best" implementation requires significant programmer effort, *e.g.* plug-in architectures, design patterns and meta-programming.

Rather than asking the programmer to build flexibility into his code, we seek a language mechanism to provide it. If we enhance the flexibility of bytecode itself, then component discovery and selection can be built into the run-time system and all suitably-compiled code can benefit from the "best" classes in the execution environment. To this end, we advocate bytecode that mentions *type variables* as well as class names, *e.g.*

```
1  new <Class A>
...
5  loadfield <Field X f>
6  loadfield <Field C g>
```

Flexible dynamic linking understands type variables in loaded classes and substitutes them by class names *as late as possible during execution.* We believe this approach, rather than substituting at install-time or program startup [ADDZ05], is in keeping with the "lazy" dynamic linking specified for the Java and .NET environments [LY99, ECM02]. It allows the choice of substitution to happen at a point in time when the most relevant information is available. Verification normally loads classes to check subtypes, but when type variables are involved, it would have to substitute them eagerly; to help defer substitutions, we allow assumptions to be made about subtypes which the verifier takes into account, following [QGC00].

We model *safe* flexible dynamic linking that results in a program where objects are accessed according to their classes. We are not concerned with *how* substitutions are chosen, but rather *when* they can be chosen and applied to a program while maintaining type-safety. Our perception of how flexible the timing of substitutions can be evolved during this work; having originally thought that type variables would have to be substituted before resolution, we discovered substitution could happen even later. The extent of type variables also evolved, from appearing in only bytecode annotations to appearing as field types and in method signatures.

The rest of this paper is structured as follows. §2 introduces a basic framework for flexible dynamic linking. §3 and §4 interleave a description of the formal system with the main design decisions about how to preserve safety (§3) and flexibility (§4). §5 formulates soundness given the decision in earlier sections. §6 mentions related work, including our current and planned implementations.

2 A language for dynamic linking

2.1 Bytecode

$$
\begin{array}{lcl}
\mathtt{CL} & ::= & \mathtt{class\ C\ extends\ C}\ \{\ \mathtt{T\ f;\ K\ \bar{M}}\ \} \\
\mathtt{K} & ::= & \mathtt{C(\bar{T}\ \bar{f})}\ \{\ \mathtt{super(\bar{f});\ this.f\ =\ f;}\ \} \\
\mathtt{M} & ::= & \mathtt{T\ m(T\ y)}\ \{\ \mathtt{return\ e;}\ \} \\
\mathtt{T} & ::= & \mathtt{C} \mid \mathtt{X} \\
\mathtt{e} & ::= & \mathtt{new}\ \mathsf{C}(\bar{\mathsf{e}})(\!|\bar{\mathsf{T}}|\!) \mid \mathsf{e.m}(\!|\mathsf{T},\mathsf{T},\mathsf{T}|\!)(\mathsf{e}) \mid \mathsf{e.f}(\!|\mathsf{T},\mathsf{T}|\!) \mid \mathsf{e.f}(\!|\mathsf{T},\mathsf{T}|\!) := \mathsf{e} \mid \mathsf{y} := \mathsf{e} \mid \mathsf{y} \mid \mathtt{this} \mid \mathtt{v} \\
\mathtt{H} & ::= & (\mathtt{this} \mapsto \iota^{+}) \cup (\mathtt{y} \mapsto \iota^{+}) \cup (\iota^{+} \mapsto (\mathtt{C} \mid \mathtt{v}))^{*} \\
\mathtt{v} & ::= & \iota \mid \mathbf{0} \mid \mathtt{linkExc} \mid \mathtt{nullPtrExc} \\
\iota & ::= & \mathbb{N}
\end{array}
$$

Fig. 1: Bytecode syntax

We consider the problem of linking and executing bytecode that is annotated with type variables and classes. In fig. 1, we define bytecode as an imperative extension of Featherweight Java [IPW99] with type annotations, similar to those in [AZ04].

Metavariable `C` ranges over class names, `X` over type variables, and `T` over types, which are classes or type variables. Each class has at most one immediate superclass, exactly one field, exactly one constructor, and zero or more methods. A constructor's formal parameters correspond to the fields inherited and declared by the class, to which the constructor assigns the parameters. Each method has a single parameter `y` and consists of a single expression. We support field hiding and method overloading and overriding.

Field access, field update, method call and object creation expressions are annotated with types (*i.e.* type variables or classes) that permit resolution of the target member. Field declarations and method signatures also contain types, rather than just classes.

Heaps map the identifiers `this` and `y` to non-zero addresses, and addresses to $\mathbf{0}$ values or class names. An object of class `C` is laid out on the heap with `C` at some address ι, followed by contiguous $\mathbf{0}$ values at $\iota+1\ldots\iota+n$ for `C`'s fields. Our heap is at a lower level than in many object-oriented models, in order to model offset calculation and demonstrate type-safety in its presence.

2.2 Programs

We consider dynamic linking to include class loading and verification. To permit verification without loading classes, we allow verification to take assumptions about subtypes into account [QGC00], and thus consider that making a subtype assumption is a linking step. Also, we consider that determining a substitution from a type variable to a class is a linking step, since it refine the types in bytecode annotations and hence allows more precise member access.

$$\begin{array}{lcl}
\mathbb{P} & ::= & (\mathbb{P}_{\mathbb{A}}, \mathbb{P}_{\mathbb{B}}, \mathbb{P}_{\mathbb{C}}, \mathbb{P}_{\mathbb{D}}) \\
\mathbb{P}_{\mathbb{A}} & ::= & (\mathtt{T}, \mathtt{T})^* \\
\mathbb{P}_{\mathbb{B}} & ::= & (\mathtt{C}, \mathtt{m}, \mathtt{T}, \mathtt{T}, \mathtt{e})^* \\
\mathbb{P}_{\mathbb{C}} & ::= & (\mathtt{C}, \mathtt{CL})^* \\
\mathbb{P}_{\mathbb{D}} & ::= & [\mathtt{T}/\mathtt{T}]^*
\end{array}$$

Fig. 2: Program syntax

To model this multi-step dynamic linking, a program $\mathbb{P}$ is a tuple as in fig. 2:

- $\mathbb{P}_{\mathbb{A}}$ stores assumptions about subtypes, where an assumption $(\mathtt{T}, \mathtt{T}')$ means "Type $\mathtt{T}$ is assumed to be a subtype of type $\mathtt{T}'$".
- $\mathbb{P}_{\mathbb{B}}$ stores verified method bodies, where ($\mathtt{C}$, $\mathtt{m}$, $\mathtt{T}_r$, $\mathtt{T}_p$, $\mathtt{e}$) means the body $\mathtt{e}$ of verified method $\mathtt{m}$ in class $\mathtt{C}$ with return type $\mathtt{T}_r$ and formal parameter type $\mathtt{T}_p$.
- $\mathbb{P}_{\mathbb{C}}$ stores class definitions, where ($\mathtt{C}$, $\mathtt{CL}_{\mathtt{C}}$) is a class definition $\mathtt{CL}_{\mathtt{C}}$ read from disk for class $\mathtt{C}$.
- $\mathbb{P}_{\mathbb{D}}$ stores determined substitutions, where $[\mathtt{C}/\mathtt{X}]$ is the determination that type variable $\mathtt{X}$ should be substituted by class $\mathtt{C}$.

2.3 The type system

(TY1)
$$\frac{}{\begin{array}{l}\mathbb{P}, \Gamma \vdash \mathtt{this} : \Gamma(\mathtt{this}) \\ \mathbb{P}, \Gamma \vdash \mathtt{y} : \Gamma(\mathtt{y}) \\ \mathbb{P}, \Gamma \vdash \mathbf{0} : \mathtt{T}\end{array}}$$

(TY2)
$$\frac{\mathbb{P}, \Gamma \vdash \bar{\mathtt{e}} : \bar{\mathtt{T}}'' \qquad \mathbb{P} \vdash_{\mathbb{A}} \bar{\mathtt{T}}'' \leq \bar{\mathtt{T}}'}{\mathbb{P}, \Gamma \vdash \mathtt{new}\ \mathtt{T}(\bar{\mathtt{e}})\llparenthesis \bar{\mathtt{T}}' \rrparenthesis : \mathtt{T}}$$

(TY3)
$$\frac{\mathbb{P}, \Gamma \vdash \mathtt{e} : \mathtt{T} \qquad \mathbb{P} \vdash_{\mathbb{A}} \mathtt{T} \leq \mathtt{T}_d}{\mathbb{P}, \Gamma \vdash \mathtt{e.f}\llparenthesis \mathtt{T}_d, \mathtt{T}_f \rrparenthesis : \mathtt{T}_f}$$

(TY4)
$$\frac{\begin{array}{ll}\mathbb{P}, \Gamma \vdash \mathtt{e} : \mathtt{T} & \mathbb{P} \vdash_{\mathbb{A}} \mathtt{T} \leq \mathtt{T}_d \\ \mathbb{P}, \Gamma \vdash \mathtt{e}' : \mathtt{T}' & \mathbb{P} \vdash_{\mathbb{A}} \mathtt{T}' \leq \mathtt{T}_p\end{array}}{\mathbb{P}, \Gamma \vdash \mathtt{e.m}\llparenthesis \mathtt{T}_d, \mathtt{T}_r, \mathtt{T}_p \rrparenthesis(\mathtt{e}') : \mathtt{T}_r}$$

(TY5)
$$\frac{\begin{array}{l}\mathbb{P}, \Gamma \vdash \mathtt{e.f}\llparenthesis \mathtt{T}_d, \mathtt{T}_f \rrparenthesis : \mathtt{T}_f \\ \mathbb{P}, \Gamma \vdash \mathtt{e}' : \mathtt{T} \qquad \mathbb{P} \vdash_{\mathbb{A}} \mathtt{T} \leq \mathtt{T}_f\end{array}}{\mathbb{P}, \Gamma \vdash \mathtt{e.f}\llparenthesis \mathtt{T}_d, \mathtt{T}_f \rrparenthesis := \mathtt{e}' : \mathtt{T}}$$

(TY6)
$$\frac{\begin{array}{l}\mathbb{P}, \Gamma \vdash \mathtt{y} : \mathtt{T}_{\mathtt{y}} \\ \mathbb{P}, \Gamma \vdash \mathtt{e} : \mathtt{T} \qquad \mathbb{P} \vdash_{\mathbb{A}} \mathtt{T} \leq \mathtt{T}_y\end{array}}{\mathbb{P}, \Gamma \vdash \mathtt{y} := \mathtt{e} : \mathtt{T}}$$

Fig. 3: Expression typing

The typing rules for expressions in fig. 3 are standard except that they use *assumptive subtyping*. Assumptive subtyping has the form $\mathbb{P} \vdash_{\mathbb{A}} \mathtt{T} \leq \mathtt{T}'$ and takes the subtype assumptions in $\mathbb{P}_{\mathbb{A}}$ into account. Thus, the typing rules can judge subtypes involving type variables (*e.g.* that class $\mathtt{C}$ is a subtype of type variable $\mathtt{X}$) and classes that have not been loaded yet.

By contrast, *definite subtyping* has the form $\mathbb{P} \vdash \mathtt{T} \leq \mathtt{T}'$ and is ordinary subclassing based on the inheritance hierarchy of loaded classes in $\mathbb{P}_{\mathbb{C}}$. Notice that subtyping rule ST5 allows an assumptive subtype to subsume a definite subtype, since it is safe to *assume* that $\mathtt{C}$ subtypes $\mathtt{C}'$ if indeed it does.

(ST1)
$$\frac{\mathbb{P}_{\mathbb{C}}(\mathtt{C}) = \mathtt{class\ C\ extends\ C'\ \{\ ...\ \}}}{\mathbb{P}\ \vdash\ \mathtt{C} \leq \mathtt{C'}}$$

(ST2)
$$\frac{}{\mathbb{P}\ \vdash\ \mathtt{T} \leq \mathtt{T}}$$

(ST3)
$$\frac{\mathbb{P}\ \vdash\ \mathtt{T''} \leq \mathtt{T'} \qquad \mathbb{P}\ \vdash\ \mathtt{T} \leq \mathtt{T''}}{\mathbb{P}\ \vdash\ \mathtt{T} \leq \mathtt{T'}}$$

(ST4)
$$\frac{(\mathtt{T}, \mathtt{T'}) \in \mathbb{P}_{\mathbb{A}}}{\mathbb{P} \vdash_{\mathbb{A}} \mathtt{T} \leq \mathtt{T'}}$$

(ST5)
$$\frac{\mathbb{P}\ \vdash\ \mathtt{T} \leq \mathtt{T'}}{\mathbb{P} \vdash_{\mathbb{A}} \mathtt{T} \leq \mathtt{T'}}$$

(ST6)
$$\frac{\mathbb{P} \vdash_{\mathbb{A}} \mathtt{T''} \leq \mathtt{T'} \qquad \mathbb{P} \vdash_{\mathbb{A}} \mathtt{T} \leq \mathtt{T''}}{\mathbb{P} \vdash_{\mathbb{A}} \mathtt{T} \leq \mathtt{T'}}$$

Fig. 4: Subtyping

2.4 Program extension

Dynamic linking modifies a program by loading and verifying classes, adding assumptions and making substitutions. To model this, we introduce the concept of *program extension*, $\vdash\ \mathbb{P}' \leq \mathbb{P}$. This judges that a program $\mathbb{P}'$ extends a program $\mathbb{P}$ via:

Assumption addition Adds a subtype assumption $(\mathtt{T}, \mathtt{T'})$ to $\mathbb{P}_{\mathbb{A}}$.
Written $\vdash\ \mathbb{P} \oplus (\mathtt{T}, \mathtt{T'}) \leq \mathbb{P}$

Class verification Using the typing rules, and thus the assumptions in $\mathbb{P}_{\mathbb{A}}$, adds a verified class $\mathtt{CV_C}$ to $\mathbb{P}_{\mathbb{B}}$.
Written $\vdash\ \mathbb{P} \oplus \mathtt{CV_C} \leq \mathbb{P}$

Class loading Adds a new class declaration $\mathtt{CL_C}$ to $\mathbb{P}_{\mathbb{C}}$.
Written $\vdash\ \mathbb{P} \oplus \mathtt{CL_C} \leq \mathbb{P}$

Substitution determination Adds a substitution σto $\mathbb{P}_{\mathbb{D}}$.
Written $\vdash\ \mathbb{P} \oplus [\sigma] \leq \mathbb{P}$

We save the definitions of these judgements until §3.2, where we discuss their role in producing and preserving a well-formed program.

3 Safety in dynamic linking

Equipped with a representation for programs whose bytecode is annotated with types, the key design decisions are:

D1 What it means for a program to be well-formed,
i.e. to guarantee type-preserving execution.

D2 How to preserve program well-formedness during linking.

D3 Where to allow type variables in bytecode annotations.

D4 When to substitute type variables.

In this section, we visit D1 and D2; the next section considers D3 and D4.

3.1 Well-formed programs (D1)

The definition of well-formedness aims to guarantee sound execution and to allow its preservation to be easily checked by program extension. We say that a program is well-formed if:

WF-Classes The loaded class hierarchy is well-formed, written $\vdash \mathbb{P}_{\mathbb{C}}$.

WF-Bodies Method bodies are well-typed with respect to the loaded class hierarchy and subtype assumptions, written $\mathbb{P}_{\mathbb{C}}, \mathbb{P}_{\mathbb{A}} \vdash \mathbb{P}_{\mathbb{B}}$.

(Notice we do not require that the assumptions in $\mathbb{P}_{\mathbb{A}}$ are well-formed in any way, such as being acyclic; we discuss this issue in §3.2.1.)

(WF-Classes)

$$\frac{\begin{array}{l}\mathbb{P}_{\mathbb{C}}(\texttt{Object}) = \texttt{class Object } \{\} \\ \mathbb{P} \vdash \texttt{C} \leq \texttt{C}' \;\wedge\; \texttt{C} \neq \texttt{C}' \implies \texttt{C}' \in dom(\mathbb{P}_{\mathbb{C}}) \\ \mathbb{P} \vdash \texttt{C} \leq \texttt{C}' \;\wedge\; \mathbb{P} \vdash \texttt{C}' \leq \texttt{C} \implies \texttt{C} = \texttt{C}'\end{array}}{\vdash \mathbb{P}_{\mathbb{C}}}$$

(WF-Bodies)

$$\frac{\begin{array}{l}\forall \texttt{C} \in dom(\mathbb{P}_{\mathbb{C}}) : \mathbb{P}_{\mathbb{B}}(\texttt{C}) \neq \epsilon \implies \\ \quad constructor(\mathbb{P}, \texttt{C}) = \bar{\texttt{U}} \;\wedge\; fields(\mathbb{P}, \texttt{C}) = \bar{\texttt{T}}\;\bar{\texttt{f}} \implies \mathbb{P} \vdash_{\mathbb{A}} \bar{\texttt{U}} \leq \bar{\texttt{T}} \\ \quad \mathbb{P}_{\mathbb{B}}(\texttt{C})(\texttt{m}, \texttt{T}_r, \texttt{T}_p) = \texttt{e} \implies \\ \quad\quad \mathbb{P}, \{\texttt{this} : \texttt{C}, \texttt{y} : \texttt{T}_p\} \vdash \texttt{e} : \texttt{T}' \;\wedge\; \mathbb{P} \vdash_{\mathbb{A}} \texttt{T}' \leq \texttt{T}_r\end{array}}{\mathbb{P}_{\mathbb{C}}, \mathbb{P}_{\mathbb{A}} \vdash \mathbb{P}_{\mathbb{B}}}$$

Fig. 5: Well-formed loaded class hierarchy and well-typed methods

The most satisfactory definition (fig. 6) would say that a program is *strongly* well-formed ($\vdash \mathbb{P}$) if there exists a program extension that gives a larger program $\mathbb{P}'$ where:

1. $\mathbb{P}'$ satisfies **WF-Classes**,
2. $\mathbb{P}'$ satisfies **WF-Bodies** without using any subtype assumptions,
3. $\mathbb{P}'$ has had all the type variables in $\mathbb{P}$ substituted.

(WF-Program-Strong)

$$\frac{\begin{array}{l}\exists \mathbb{P}' : \\ \quad \vdash \mathbb{P}' \leq \mathbb{P} \\ \quad \vdash \mathbb{P}_{\mathbb{C}}' \qquad \mathbb{P}_{\mathbb{C}}', \emptyset \vdash \mathbb{P}_{\mathbb{B}}' \\ \quad \mathbb{P}' \textit{ does not contain type variables}\end{array}}{\vdash \mathbb{P}}$$

Fig. 6: Strongly well-formed programs

While the definition of well-formedness in fig. 6 would be the most desirable, as it would detect errors as early as possible, it is difficult to check its preservation because arbitrary program extension is required. We therefore weaken the definition to that in fig. 7, allowing the program to contain both assumptions and type variables:

$$\boxed{\begin{array}{l}\text{(WF-Program-Global)}\\ \dfrac{\vdash\ \mathbb{P}_{\mathbb{C}} \qquad \mathbb{P}_{\mathbb{C}}, \mathbb{P}_{\mathbb{A}}\ \vdash\ \mathbb{P}_{\mathbb{B}}}{\vdash\ \mathbb{P}}\end{array}}$$

Fig. 7: Well-formed programs

Preserving **WF-Classes** is straightforward: when a new class is loaded, we just require that its superclasses are already loaded and it must not introduce a cycle into the class hierarchy. However, a newly loaded class introduces subtypes that may conflict with assumptions already used to verify a method body. For example, verification of `this .f⦇A,B⦈ = new C()⦇⦈` may depend on the assumption `(C,B)`; if `C` is loaded and does not subtype `B`, then type-safety is lost. Therefore, preserving $\mathbb{P}_{\mathbb{C}}', \emptyset\ \vdash\ \mathbb{P}_{\mathbb{B}}'$ would require re-verifying existing method bodies at class loading, to check subtypes involving the newly loaded class.

It would also mean re-verifying method bodies when a substitution is made, because substitutions can change subtype assumptions. For example, verification of `this .f⦇A,X⦈ = new C()⦇⦈` depends on the assumption (`C`,`X`). If we substitute `X` by `B`, then we would need to re-verify the expression under the assumption `(C,B)`.

Because the definition in fig. 7 requires global checks, we add a third property to well-formed programs to accompany **WF-Classes** and **WF-Bodies**:

WF-Assumptions Subtype assumptions do not contradict the loaded class hierarchy, written $\mathbb{P}_{\mathbb{C}}\ \vdash\ \mathbb{P}_{\mathbb{A}}$.

We augment the definition of a well-formed program from fig. 7 to obtain fig. 8.

$$\boxed{\begin{array}{l}\text{(WF-Program)}\\ \dfrac{\vdash\ \mathbb{P}_{\mathbb{C}} \qquad \mathbb{P}_{\mathbb{C}}, \mathbb{P}_{\mathbb{A}}\ \vdash\ \mathbb{P}_{\mathbb{B}} \qquad \mathbb{P}_{\mathbb{C}}\ \vdash\ \mathbb{P}_{\mathbb{A}}}{\vdash\ \mathbb{P}}\end{array}}$$

Fig. 8: Well-formed programs

A possible definition of **WF-Assumptions** would be that any assumptions made to support verification are either true in the loaded class hierarchy or will eventually become true when further classes are loaded, modulo substitution of the assumption. Thus, with respect to the classes and assumptions in a well-formed program, there should exist a larger set of classes and substitutions such that any subtype assumed to hold in the original program also holds, with the substitutions applied, in the larger program, as in fig. 9. (We write $\mathbb{P}_{\mathbb{D}}(\mathtt{T})$ to signify the substitution of `T` according to $\mathbb{P}_{\mathbb{D}}$.)

$$\boxed{\begin{array}{l}\text{(WF-Assumptions-Strong)}\\ \exists\mathbb{P}' : \\ \dfrac{\begin{array}{l}\qquad \vdash\ \mathbb{P}' \leq \mathbb{P}\\ \qquad \mathbb{P} \vdash_{\mathbb{A}} \mathtt{T} \leq \mathtt{T}' \implies \mathbb{P}'\ \vdash\ (\mathbb{P}')_{\mathbb{D}}(\mathtt{T}) \leq (\mathbb{P}')_{\mathbb{D}}(\mathtt{T}')\end{array}}{\mathbb{P}_{\mathbb{C}}\ \vdash\ \mathbb{P}_{\mathbb{A}}}\end{array}}$$

Fig. 9: Strongly well-formed assumptions

However, this formulation of **WF-Assumptions** is too strong, because it is possible to make assumptions about non-existent classes such that *no* larger program can be found to satisfy the assumptions. It is also hard to implement, because there is no obvious mechanism for finding the larger program.

Therefore, we require a weaker formulation of **WF-Assumptions**, namely that an assumption about a class's superclass is true iff the class is loaded in the current program. We show this formulation in 10; it is weaker than fig. 9 because it talks about classes only, not types. Therefore, substitutions do not affect it, since applying a substitution to a class just returns the class.

$$\boxed{\begin{array}{l}\text{(WF-ASSUMPTIONS)}\\ \dfrac{\mathbb{P} \vdash_{\mathbb{A}} \mathtt{C} \leq \mathtt{C}' \;\wedge\; \mathtt{C} \in \mathit{dom}(\mathbb{P}_{\mathbb{C}}) \;\Longrightarrow\; \mathbb{P} \vdash \mathtt{C} \leq \mathtt{C}'}{\mathbb{P}_{\mathbb{C}} \vdash \mathbb{P}_{\mathbb{A}}}\end{array}}$$

Fig. 10: Well-formed assumptions

3.2 Checking well-formedness (D2)

We rely on the program extension rules defined in fig. 11 (and first mentioned in §2.4) to preserve program well-formedness. Rule PE1 loads a class, PE2 verifies a method body, PE3 and PE4 add an assumption, and PE5 adds a substitution that has been determined. We do *not* describe how an external agent determines a substitution, or chooses which class to load or verify, or assumption to add.

The $\oplus$ operator adds to a program $\mathbb{P} = (\mathbb{P}_{\mathbb{A}},\mathbb{P}_{\mathbb{B}},\mathbb{P}_{\mathbb{C}},\mathbb{P}_{\mathbb{D}})$ in a straightforward way:

$$\begin{array}{lcl}
\mathbb{P} \oplus (\mathtt{T},\mathtt{T}') & = & ((\mathbb{P}_{\mathbb{A}},\, (\mathtt{T},\mathtt{T}')),\, \mathbb{P}_{\mathbb{B}},\, \mathbb{P}_{\mathbb{C}},\, \mathbb{P}_{\mathbb{D}})\\
\mathbb{P} \oplus \mathtt{CV}_{\mathtt{C}} & = & (\mathbb{P}_{\mathbb{A}},\, (\mathbb{P}_{\mathbb{B}},\, (\mathtt{C},\, \mathtt{CV}_{\mathtt{C}})),\, \mathbb{P}_{\mathbb{C}},\, \mathbb{P}_{\mathbb{D}})\\
\mathbb{P} \oplus \mathtt{CL}_{\mathtt{C}} & = & (\mathbb{P}_{\mathbb{A}},\, \mathbb{P}_{\mathbb{B}},\, (\mathbb{P}_{\mathbb{C}},\, (\mathtt{C},\mathtt{CL}_{\mathtt{C}})),\, \mathbb{P}_{\mathbb{D}})\\
\mathbb{P} \oplus [\sigma] & = & (\mathbb{P}_{\mathbb{A}}\, [\sigma],\, \mathbb{P}_{\mathbb{B}}\, [\sigma],\, \mathbb{P}_{\mathbb{C}}\, [\sigma],\, (\mathbb{P}_{\mathbb{D}},\, [\sigma]))
\end{array}$$

(We discuss why adding a substitution has the side-effect of applying the substitution throughout the program in §4.3.)

The following table summarises which program extension rules preserve which aspects of well-formedness:

Program extension	Notation	Modifies	Preserves
Class loading (PE1)	$\vdash\ \mathbb{P} \oplus \mathtt{CL}_{\mathtt{C}} \leq \mathbb{P}$	$\mathbb{P}_{\mathbb{C}}$	$\mathbb{P}_{\mathbb{C}} \vdash \mathbb{P}_{\mathbb{A}},\ \vdash \mathbb{P}_{\mathbb{C}}$
Class verification (PE2)	$\vdash\ \mathbb{P} \oplus \mathtt{CV}_{\mathtt{C}} \leq \mathbb{P}$	$\mathbb{P}_{\mathbb{B}}$	$\mathbb{P}_{\mathbb{C}},\mathbb{P}_{\mathbb{A}} \vdash \mathbb{P}_{\mathbb{B}}$
Assumption addition (PE3,4)	$\vdash\ \mathbb{P} \oplus (\mathtt{T},\mathtt{T}') \leq \mathbb{P}$	$\mathbb{P}_{\mathbb{A}}$	$\mathbb{P}_{\mathbb{C}} \vdash \mathbb{P}_{\mathbb{A}}$
Substitution determination (PE5)	$\vdash\ \mathbb{P} \oplus [\sigma] \leq \mathbb{P}$	$\mathbb{P}_{\mathbb{D}}$	$\mathbb{P}_{\mathbb{C}} \vdash \mathbb{P}_{\mathbb{A}}$

$\vdash \mathbb{P}_{\mathbb{C}}$ is preserved by rule PE1 requiring that the superclass of the class being loaded is already loaded. $\mathbb{P}_{\mathbb{C}},\mathbb{P}_{\mathbb{A}} \vdash \mathbb{P}_{\mathbb{B}}$ is preserved by rule PE2 typing a class's constructor and methods using the typing rules from fig. 3, and then creating the class's virtual table with the method addition operator, $\bullet$. This operator (defined in the appendix) adds methods to $\mathbb{P}_{\mathbb{B}}$ and supports overriding by preferring to add a method from a newly verified class rather than its superclass.

The most interesting preservation is of $\mathbb{P}_{\mathbb{C}} \vdash \mathbb{P}_{\mathbb{A}}$, to which three program extension rules contribute:

Rule PE1 requires that any classes that are assumptive supertypes of the newly loaded class, are also its definite supertypes.

(PE1)

$$\frac{\begin{array}{l}\mathtt{CL_C} = \mathtt{class\ C\ extends\ C'\ \{\ T\ f;\ K\ \bar{M}\ \}} \\ \mathbb{P}_{\mathbb{B}}(\mathtt{C}) = \epsilon \qquad \mathbb{P}_{\mathbb{C}}(\mathtt{C}) = \epsilon \qquad \mathbb{P}_{\mathbb{C}}(\mathtt{C'}) \neq \epsilon \\ \mathbb{P} \vdash_{\mathbb{A}} \mathtt{C} \leq \mathtt{C''} \implies (\mathtt{C''} = \mathtt{C'} \ \vee\ \mathbb{P} \vdash \mathtt{C'} \leq \mathtt{C''})\end{array}}{\vdash\ \mathbb{P} \oplus \mathtt{CL_C} \leq \mathbb{P}}$$

(PE2)

$$\frac{\begin{array}{l}\mathbb{P}_{\mathbb{C}}(\mathtt{C}) = \mathtt{class\ C\ extends\ C'\ \{\ T\ f;\ K\ \bar{M}\ \}} \\ \mathbb{P}_{\mathbb{B}}(\mathtt{C}) = \epsilon \qquad \mathbb{P}_{\mathbb{B}}(\mathtt{C'}) \neq \epsilon \\ \mathit{constructor}(\mathbb{P}, \mathtt{C'}) = \bar{\mathtt{U}} \\ \mathtt{K} = \mathtt{C(\bar{U}\ \bar{g},\ V\ f)\ \{\ super(\bar{g});\ this.f\ =\ f;\ \}} \implies \mathbb{P} \vdash_{\mathbb{A}} \mathtt{V} \leq \mathtt{T} \\ \forall i : 1..n \quad \mathtt{M}^i = \mathtt{T_r}^i\ \mathtt{m}^i(\mathtt{T_p}^i\ \mathtt{y}^i)\ \{\ \mathtt{return\ e}^i;\ \} \implies \\ \qquad\qquad \mathbb{P}, \{\mathtt{this} : \mathtt{C}, \mathtt{y}^i : \mathtt{T_p}^i\} \vdash \mathtt{e}^i : \mathtt{T}^i \ \wedge\ \mathbb{P} \vdash_{\mathbb{A}} \mathtt{T}^i \leq \mathtt{T_r}^i \\ \mathtt{CV_C} = \mathbb{P}_{\mathbb{B}}(\mathtt{C'}) \bullet (\mathtt{m}^i, \mathtt{T_r}^i, \mathtt{T_p}^i \mapsto \mathtt{e}^i)^{i=1..n}\end{array}}{\vdash\ \mathbb{P} \oplus \mathtt{CV_C} \leq \mathbb{P}}$$

(PE3)

$$\frac{\begin{array}{l}\mathtt{C} \in \mathit{dom}(\mathbb{P}_{\mathbb{C}}) \ \wedge\ \mathtt{T} \in \mathtt{Classes} \implies \mathbb{P} \vdash \mathtt{C} \leq \mathtt{T} \\ \mathtt{C} \in \mathit{dom}(\mathbb{P}_{\mathbb{C}}) \ \wedge\ \mathtt{T} \in \mathtt{TypeVars} \ \wedge\ \mathbb{P} \vdash_{\mathbb{A}} \mathtt{T} \leq \mathtt{C'} \implies \mathbb{P} \vdash \mathtt{C} \leq \mathtt{C'}\end{array}}{\vdash\ \mathbb{P} \oplus (\mathtt{C}, \mathtt{T}) \leq \mathbb{P}}$$

(PE4)

$$\frac{\mathbb{P} \vdash_{\mathbb{A}} \mathtt{C} \leq \mathtt{X} \ \wedge\ \mathtt{C} \in \mathit{dom}(\mathbb{P}_{\mathbb{C}}) \ \wedge\ \mathtt{T} \in \mathtt{Classes} \implies \mathbb{P} \vdash \mathtt{C} \leq \mathtt{T}}{\vdash\ \mathbb{P} \oplus (\mathtt{X}, \mathtt{T}) \leq \mathbb{P}}$$

(PE5) $$\frac{\mathbb{P} \vdash \sigma \diamond}{\vdash\ \mathbb{P} \oplus [\sigma] \leq \mathbb{P}}$$

(PE6) $$\frac{\vdash\ \mathbb{P}' \leq \mathbb{P}'' \qquad \vdash\ \mathbb{P}'' \leq \mathbb{P}}{\vdash\ \mathbb{P}' \leq \mathbb{P}}$$

(PE7) $$\frac{}{\vdash\ \mathbb{P} \leq \mathbb{P}}$$

Fig. 11: Program extension

Rule PE3 requires that, when adding an assumption that class `C` is a definite subtype of another class, the assumption is confirmed by the class hierarchy *if* `C` *is already loaded.*

Rule PE5 requires that assumptions are still confirmed by the loaded class hierarchy after a well-formed substitution σ is determined, since a substitution applies to the whole program including assumptions. Well-formed substitutions are defined in fig. 12. Namely, to determine that [`C`/`X`] is a valid substitution, we require that if `C` is loaded, then `C`'s superclasses must match those assumed to be `X`'s supertypes; also, any loaded class which was assumed to subtype `X` must also subtype `C`.

(WFS1)

$$\frac{\begin{array}{l}\mathtt{C} \in \mathit{dom}(\mathbb{P}_{\mathbb{C}}) \ \wedge\ \mathbb{P} \vdash_{\mathbb{A}} \mathtt{X} \leq \mathtt{C'} \implies \mathbb{P} \vdash \mathtt{C} \leq \mathtt{C'} \\ \mathtt{C'} \in \mathit{dom}(\mathbb{P}_{\mathbb{C}}) \ \wedge\ \mathbb{P} \vdash_{\mathbb{A}} \mathtt{C'} \leq \mathtt{X} \implies \mathbb{P} \vdash \mathtt{C'} \leq \mathtt{C}\end{array}}{\mathbb{P} \vdash \mathtt{C/X} \diamond}$$

(WFS2)

$$\frac{}{\mathbb{P} \vdash \mathtt{T/T} \diamond}$$

Fig. 12: Well-formed substitutions

3.2.1 Cyclic assumptions

Cyclic assumptions do not interfere with soundness but can prevent a class from being loaded. For example, if $\mathbb{P}_\mathbb{A}$ contains assumptions (A,B) and (B,A), and neither A nor B is loaded in $\mathbb{P}_\mathbb{C}$, then neither class can *ever* be loaded. In rule PE1, loading A requires that the assumption that it subtypes B is confirmed by the loaded class hierarchy, which (in a well-formed program) will require B to be loaded. However, at the point of loading B, we had to check the assumption that it subtypes A, requiring A to be loaded. Thus, with the cycle between A and B already in the assumptions, loading A requires B already loaded, and loading B requires A already loaded, so neither class can be loaded.[1]

For this reason, we considered forbidding cyclic assumptions. It would be straightforward to augment rule PE3 to only allow assumption $(\mathtt{C},\mathtt{T})$ iff it does not form a cycle:

$\mathbb{P} \vdash_\mathbb{A} \mathtt{T} \leq \mathtt{C}' \quad \Longrightarrow \quad \mathbb{P} \not\vdash_\mathbb{A} \mathtt{C}' \leq \mathtt{C}$

However, assumptions are affected by substitutions, and the logic for checking that a substitution does not introduce a cyclic assumption is complicated. It depends on what classes are loaded, what assumptions exist and which of those assumptions will be affected by the substitution. To keep a simple presentation, we decided to allow cycles in assumptions.

4 Flexibility in dynamic linking

4.1 Location of type variables (D3)

A contributor to sound execution is that objects can only be created at classes, but fields can be declared and accessed at types. Thus, we know that any member access - even annotated with type variables - actually points to an instance of a loaded class. As an object passes from one field or variable to another (namely, when a reference is passed as a parameter or assigned to a field), successful verification will require assumptions that describe subtype relationships for the types of those fields and variables. As discussed in §3, if the assumptions involve classes, then it is not possible for those classes to "escape" having the assumptions checked; this either happens when an assumption is made (if the class is loaded at the time) or when the class is loaded (if not). If an assumption relates a class to a type variable, *e.g.* (C,X), then it is checked when the type variable is substituted.

For example, suppose we create an object of class C and assign it to a field of type X. Verification requires the assumption that C subtypes X. Elsewhere, suppose there is an assignment of an expression of type X to a field of type D. Verification requires the assumption that X subtypes D. Any eventual substitution for X must respect these assumptions, so that whatever class substitutes for X is at least a D and at most a C.

Each type variable may be substituted independently, but there may be some type variables used in assumptions that are never substituted. Even in this case, we still maintain soundness. As an object is passed around, a chain of subtype assumptions involving type variables is built for verification. As soon as verification sees an object pass from an expression at a type variable to an expression at a class, the type variables in the assumption chain "collapse", leaving a checkable assumption relating

[1] On the other hand, once a class is loaded, then no cycle of assumptions involving that class can be created, by rule PE3.

the object's declared class to the final expression's class.

For example, suppose we create an object of class `C` and pass it into a formal parameter of type `X`. Suppose also that the parameter is assigned to a field declared at type `Y`, and that field has a method invoked on it from class `D`. For verification to succeed on this code, we require assumptions that `C` subtypes `X`, `X` subtypes `Y`, and `Y` subtypes `D`. We can always assume (`C`,`X`) and (`X`,`Y`) by rule PE4, but then two scenarios are possible:

1. If class `C` is already loaded before the `C` object creation is verified, then in order to make the assumption that `Y` subtypes `D` - and thus, transitively, that `C` subtypes `D` - *it must actually be the case that* `C` *subtypes* `D`.
2. If class `C` is not loaded when the `C` object creation is verified, then we may freely assume that `D` is a supertype. When class `C` is loaded - which happens *at the latest* when the `C` object is created - we require that `C` *indeed subtypes* `D`, thus ensuring that the sequence of assignment and method call operations above is sound.

Notice that `X` and `Y` do not need to be substituted during the sequence of operations. If they are substituted, then given the assumptions we mentioned, they must be substitute by classes that are supertypes of `C` and subtypes of `D`.

4.2 Operational semantics

Execution has the form $\mathbb{P}, \mathtt{H}, \mathtt{e} \ \leadsto\ \mathbb{P}', \mathtt{H}', \mathtt{v}$ and is defined in fig. 13. We take a non-determistic approach that allows either an expression to be executed or a link exception (`linkExc`) to be thrown at any time. Exceptions are propagated using the contexts defined in the appendix. We aim to allow types, rather than classes, to feature as widely as possible in annotations. This freedom manifests itself in the following ways:

Rule IMEX1 Field access requires that the field's defining type must be a class, but that the field itself can be a type variable or class. This is because the $offset()$ function looks up the field's defining type in the loaded classes $\mathbb{P}_{\mathbb{C}}$.

Rule IMEX3 Method call permits annotations at types rather than classes throughout. The dynamic type of the receiver, rather than the defining type of the method, is used to extract a (verified) method body from $\mathbb{P}_{\mathbb{B}}$. Therefore, since the defining type in the method call annotation is not used, we permit it to be a type, rather than a class. The dynamic type is always a class because only class instances can be laid out on the heap. The formal parameter type and return type are both used to resolve the method, allowing invocation of methods with type variables in their signatures.

Rule IMEX4 An object must be created at a class, as described above, but the annotation for an object creation expression can feature types to help resolve the class's constructor, which permits types in its signature. A new object's fields are initialised to **0**. Only verified classes ($\mathbb{P}_{\mathbb{B}}(\mathtt{C}) \neq \epsilon$) can be instantiated, as in Java. An address ι is *fresh* in heap `H` iff $\forall\kappa$: $\mathtt{H}(\iota + \kappa) = \epsilon$.

Dynamic linking can also happen at any time, through rule IMEX11. It first performs program extension and then continues execution with the larger program. In case a substitution was added to the program by program extension, we apply all known substitutions (from $\mathbb{P}_{\mathbb{D}}{}'$) to the expression before evaluating it.

(IMEX1)
$$\frac{\mathbb{P},\mathtt{H},\mathtt{e} \leadsto \mathbb{P}',\mathtt{H}',\iota_0 \qquad \kappa = \mathit{offset}(\mathbb{P}',\mathtt{f},\mathtt{C}_d,\mathtt{T}_f)}{\mathbb{P},\mathtt{H},\mathtt{e.f}(\!|\mathtt{C}_d,\mathtt{T}_f|\!) \leadsto \mathbb{P}',\mathtt{H}',\mathtt{H}'(\iota_0+\kappa)}$$

(IMEX2)
$$\frac{\mathbb{P},\mathtt{H},\mathtt{e} \leadsto \mathbb{P}',\mathtt{H}',\iota_0 \qquad \mathbb{P}',\mathtt{H}',\mathtt{e}' \leadsto \mathbb{P}'',\mathtt{H}'',\mathtt{v} \qquad \kappa = \mathit{offset}(\mathbb{P}'',\mathtt{f},\mathtt{C}_d,\mathtt{T}_f)}{\mathbb{P},\mathtt{H},\mathtt{e.f}(\!|\mathtt{C}_d,\mathtt{T}_f|\!) := \mathtt{e}' \leadsto \mathbb{P}'',\mathtt{H}''[\iota_0+\kappa \mapsto \mathtt{v}],\mathtt{v}}$$

(IMEX3)
$$\frac{\begin{array}{l}\mathbb{P},\mathtt{H},\mathtt{e} \leadsto \mathbb{P}',\mathtt{H}',\iota_0 \qquad \mathbb{P}',\mathtt{H}',\mathtt{d} \leadsto \mathbb{P}'',\mathtt{H}'',\mathtt{v}\\ \mathtt{H}''(\iota_0)=\mathtt{C} \qquad \mathit{body}(\mathbb{P}'',\mathtt{C},\mathtt{m},\mathtt{T}_r,\mathtt{T}_p)=\mathtt{e}_0\\ \mathbb{P}'',\mathtt{H}''[\mathtt{this}\mapsto\iota_0][\mathtt{y}\mapsto\mathtt{v}],\mathtt{e}_0 \leadsto \mathbb{P}''',\mathtt{H}''',\mathtt{v}'\end{array}}{\mathbb{P},\mathtt{H},\mathtt{e.m}(\!|\mathtt{T}_d,\mathtt{T}_r,\mathtt{T}_p|\!)(\mathtt{d}) \leadsto \mathbb{P}''',\mathtt{H}'''[\mathtt{this}\mapsto\mathtt{H}(\mathtt{this})][\mathtt{y}\mapsto\mathtt{H}(\mathtt{y})],\mathtt{v}'}$$

(IMEX4)
$$\frac{\begin{array}{l}\mathit{constructor}(\mathbb{P},\mathtt{C})=\bar{\mathtt{T}} \qquad \mathbb{P}_{\mathbb{B}}(\mathtt{C})\neq\epsilon\\ \mathbb{P},\mathtt{H},\mathtt{e}_1 \leadsto \mathbb{P}_1,\mathtt{H}_1,\mathtt{v}_1 \ \ \ldots\ \ \mathbb{P}_{n-1},\mathtt{H}_{n-1},\mathtt{e}_n \leadsto \mathbb{P}_n,\mathtt{H}_n,\mathtt{v}_n\\ \iota_0 \ \mathit{fresh\ in}\ \mathtt{H}_n\end{array}}{\mathbb{P},\mathtt{H},\mathtt{new\ C}(\bar{\mathtt{e}})(\!|\bar{\mathtt{T}}|\!) \leadsto \mathbb{P}_n,\mathtt{H}_n[\iota_0\mapsto\mathtt{C}][\iota_0+1\mapsto\mathtt{v}_1]\ldots[\iota_0+n\mapsto\mathtt{v}_n],\iota_0}$$

(IMEX5)
$$\frac{\mathbb{P},\mathtt{H},\mathtt{e} \leadsto \mathbb{P}',\mathtt{H}',\mathtt{v}}{\mathbb{P},\mathtt{H},\mathtt{y} := \mathtt{e} \leadsto \mathbb{P}',\mathtt{H}'[\mathtt{y}\mapsto\mathtt{v}],\mathtt{v}}$$

(IMEX6)
$$\frac{}{\begin{array}{l}\mathbb{P},\mathtt{H},\mathtt{this} \leadsto \mathbb{P},\mathtt{H},\mathtt{H}(\mathtt{this})\\ \mathbb{P},\mathtt{H},\mathtt{y} \leadsto \mathbb{P},\mathtt{H},\mathtt{H}(\mathtt{y})\\ \mathbb{P},\mathtt{H},\mathbf{0} \leadsto \mathbb{P},\mathtt{H},\mathbf{0}\end{array}}$$

(IMEX7)
$$\frac{}{\mathbb{P},\mathtt{H},\mathtt{e} \leadsto \mathbb{P},\mathtt{H},\mathtt{linkExc}}$$

(IMEX8)
$$\frac{\mathbb{P},\mathtt{H},\mathtt{e} \leadsto \mathbb{P}',\mathtt{H}',\mathbf{0}}{\mathbb{P},\mathtt{H},\sqsubset \mathtt{e} \sqsupset^{null} \leadsto \mathbb{P},\mathtt{H},\mathtt{nullPtrExc}}$$

(IMEX9)
$$\frac{\mathtt{v} \in \{\mathtt{linkExc},\mathtt{nullPtrExc}\}}{\mathbb{P},\mathtt{H},\sqsubset \mathtt{v} \sqsupset^{exe} \leadsto \mathbb{P},\mathtt{H},\mathtt{v}}$$

(IMEX10)
$$\frac{\begin{array}{l}\mathbb{P},\mathtt{H},\mathtt{e}_1 \leadsto \mathbb{P}',\mathtt{H}',\mathtt{v}_1\\ \mathbb{P}',\mathtt{H}',\mathtt{e}_2 \leadsto \mathbb{P}'',\mathtt{H}'',\mathtt{v}_2\end{array}}{\mathbb{P},\mathtt{H},\mathtt{e}_1;\mathtt{e}_2 \leadsto \mathbb{P}'',\mathtt{H}'',\mathtt{v}_2}$$

(IMEX11)
$$\frac{\vdash \mathbb{P}' \leq \mathbb{P} \qquad \mathbb{P}',\mathtt{H},(\mathbb{P}')_{\mathbb{D}}(\mathtt{e}) \leadsto \mathbb{P}'',\mathtt{H}',\mathtt{v}}{\mathbb{P},\mathtt{H},\mathtt{e} \leadsto \mathbb{P}'',\mathtt{H}',\mathtt{v}}$$

Fig. 13: Execution

4.3 Timing of substitutions (D4)

We allow execution of an expression to begin if some type variables remain in the expression's type annotation. For example, while evaluating the receiver of a method call, new information might come to light that gives a substitution for the method's formal parameter type, and thus helps resolve the method.

The only place where a class name must appear before an expression is evaluated is instantiation. We permit typing of bytecode that contains object creation expressions of the form `new T`, since we wish to be as flexible as possible in what we accept in "raw" bytecode loaded from disk. The typing rules in fig. 3 support verification of such an expression, but as discussed above and seen in rule IMEX4, we only permit objects to be created at classes. Therefore, a substitution must be applied to ensure an object creation expression of the form `new C` before it can be executed.

Our choice of large-step semantics makes applying substitutions interesting. Unless each premise of an execution rule applied substitutions explicitly to its expression, a premise could only evaluate terms exactly as they appear in the rule's conclusion. Since we wish to let premises benefit from substitutions made by earlier premises in the same rule, it is essential to store substitutions for later application. Rule IMEX11 allows any execution rule to apply existing substitutions to the current expression by means of an idempotent program extension, freeing rules IMEX1-10 from applying substitutions explicitly. The $offset()$ and $body()$ functions that perform field and method resolution in rules IMEX1-3 also apply existing substitutions.

We originally believed that applying substitutions globally throughout the program, without storing them, would be sufficient. This is only suitable for a small-step semantics, where one part of an expression can be evaluated and substitutions determined, then later parts of the same expression can be retrieved *from the program with substitutions applied* and evaluated.

5 Soundness

5.1 Formulation of soundness

A "strong" formulation of soundness is that an expression annotated with classes evaluates to a value whose type is a definite subtype of the original expression's type. However, this does not guarantee anything about expressions annotated with type variables. To guarantee sound execution, an expression would have to be fully substituted (*i.e.* no type variables remaining in its annotations) *before* it is executed, whereas we have aimed to allow substitutions *during* an expression's execution.

Therefore, we state a more general soundness property about execution of expressions annotated with type variables as well as classes. The cost of this generality is that we lose the ability to use definite subtypes; namely, executing an expression of type `T` will give a value whose type is an assumptive subtype of `T`. In fact, since substitutions can be made during an expression's execution, the value's type must be assumed to subtype *a substitution of* `T`.

5.2 Run-time typing

Our operational semantics allows us to type executable expressions using the typing rules in fig. 3. This is because executable expressions understood by fig. 13 are in the same format as expressions loaded from disk with the syntax from fig. 1; there is no intermediate rewriting of expressions into another format, *e.g.* featuring offsets. In addition, reusing the typing rules is consistent with our soundness property, since the rules use assumptive subtypes and we only require assumptive subtyping for soundness.

(HC1)

$$\frac{\begin{array}{l}\mathtt{H}(\iota) = \mathtt{C} \implies \mathbb{P},\mathtt{H} \vdash \iota \\ \mathbb{P},\mathtt{H} \vdash \mathtt{H}(\mathtt{this}) \lhd \Gamma(\mathtt{this}) \\ \mathbb{P},\mathtt{H} \vdash \mathtt{H}(\mathtt{y}) \lhd \Gamma(\mathtt{y})\end{array}}{\mathbb{P},\Gamma \vdash \mathtt{H}}$$

(HC2)

$$\frac{\begin{array}{l}\mathtt{H}(\iota) = \mathtt{C} \qquad \mathtt{C} \in \mathit{dom}(\mathbb{P}_{\mathtt{C}}) \\ \mathit{fields}(\mathbb{P},\mathtt{C}) = \mathtt{T}_1\ \mathtt{f}_1 \ldots \mathtt{T}_n\ \mathtt{f}_n \implies \\ \qquad \forall \kappa\ \mathit{in}\ 1..n : \mathtt{H}(\iota+\kappa) \neq \mathtt{T} \ \wedge\ \mathbb{P},\mathtt{H} \vdash \mathtt{H}(\iota+\kappa) \lhd \mathtt{T}_\kappa\end{array}}{\mathbb{P},\mathtt{H} \vdash \iota}$$

(HC3A)

$$\frac{\mathtt{H}(\iota) = \mathtt{C}' \qquad \mathbb{P} \vdash_{\mathbb{A}} \mathtt{C}' \leq \mathtt{T}}{\mathbb{P},\mathtt{H} \vdash \iota \lhd \mathtt{T}}$$

(HC3Z)

$$\frac{}{\mathbb{P},\mathtt{H} \vdash \mathbf{0} \lhd \mathtt{T}}$$

Fig. 14: Conformance

5.3 Conformance

Heap conformance ($\mathbb{P},\Gamma \vdash \mathtt{H}$) guarantees that the heap conforms to the program, by requiring that every object on the heap (signified by a class name at a heap location) comprises a sequence of values whose types correspond to the fields in the class to which the object belongs. Objects must not overlap on the heap, and must belong to loaded classes only ($\mathbb{P},\mathtt{H} \vdash \iota$).

The use of assumptions for typing executable expressions must be balanced by the use of assumptions to type values on the heap. Therefore, a heap address ι conforms to class `C` if the address points to an object of class `D`, and `D` is *assumed* to subtype `C`.

Additionally, because we allow fields declared at types, heap locations are actually judged with types, not classes ($\mathbb{P},\mathtt{H} \vdash \iota \lhd \mathtt{T}$). $\mathbf{0}$ is the only conformant value for a field at a type variable. `this` and `y` are also judged at types, although $\Gamma(\mathtt{this})$ will always be a class since, during verification, method bodies are typed with `this` as the defining class.

Theorem 1 (Soundness)

If

$$\begin{array}{l}\vdash \mathbb{P} \quad \text{and} \quad \mathbb{P},\Gamma \vdash \mathtt{H} \quad \text{and} \quad \mathbb{P},\Gamma \vdash \mathtt{e} : \mathtt{T} \quad \text{and} \\ \mathbb{P},\mathtt{H},\mathtt{e} \leadsto \mathbb{P}',\mathtt{H}',\mathtt{v} \quad \text{and} \quad \mathtt{v} \notin \{\mathtt{linkExc}, \mathtt{nullPtrExc}\}\end{array}$$

Then

$$\vdash \mathbb{P}' \quad \text{and} \quad \mathbb{P}', (\mathbb{P}')_{\mathbb{D}}(\Gamma) \vdash \mathtt{H}' \quad \text{and} \quad \mathbb{P}',\mathtt{H}' \vdash \mathtt{v} \lhd (\mathbb{P}')_{\mathbb{D}}(\mathtt{T})$$

Notice that the final type environment and value type both have substitutions applied, *i.e.* $(\mathbb{P}')_{\mathbb{D}}(\ldots)$. This reflects that fact that, during execution, substitutions may be applied to the original expression, potentially changing its type. Detailed proofs may be found at `http://www.doc.ic.ac.uk/~abuckley/fdl`.

6 Related and further work

Dynamic linking [Fra97] attracts formal study because of its impact on security [Dea97, JLT98, LB98] and its perceived complexity [DE02, EJS02]. There are numerous models of dynamic linking for Java [QGC00, Dro00], while [DLE03] gives a non-deterministic model that can describe linking in both Java and .NET. [HWC00] considers safe dynamic linking of native code while [AFZ03] takes a language-independent view of dynamic linking.

These models do not consider truly flexible dynamic linking with type variables. Recent advances in *compositional compilation* [AZ04] emit *polymorphic bytecode* [ADDZ05] with type variables. In [BD04], we unified a model of dynamic linking with bytecode that contains type variables, allowing very late binding to components. The imperative execution system presented in this paper develops the functional system of [BD04]. More abstract than our bytecode is the *choice operator* for dynamically choosing an implementation and linking it to an interface [AGW04].

We have a basic implementation of flexible dynamic linking in the .NET environment, titled "Flexible Dynamic Rotor" [BMED04]. In .NET's bytecode, member accesses are annotated with both the class that defines the member and the assembly (`.dll` file) that defines the class. We modified the "shared source" version of .NET's execution engine [Mic02] to interpret type variables in method call instructions, for both class and assembly annotations. The linking phases in .NET are different from those presented here (which we modelled after the JVM); the latest possible moment to perform substitution is at verification, immediately before JIT-compilation to x86 assembly code.

Further work concerns *how* substitutions are chosen. [ADDZ05] equips bytecode not only with type variables, but also with constraints about which members are expected in which class. We plan a model that matches the constraints from [ADDZ05] against the available resources in an execution environment. We believe that such a model can usefully describe the emerging world of service-oriented architectures [Kre01, Dod04], where "ultra-late binding" between Web Services [TBB03] is a core concept. We will extend our .NET implementation to allow type variables to be represented in C# source code, via custom attributes, and also plan a JVM-based implementation.

References

[ADDZ05] Davide Ancona, Ferruccio Damiani, Sophia Drossopoulou, and Elena Zucca. Polymorphic Bytecode: Compositional Compilation for Java-like Languages. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'05), to appear*, Long Beach, CA, USA, January 2005.

[AFZ03] Davide Ancona, Sonia Fagorzi, and Elena Zucca. A Calculus for Dynamic Linking. In *Proceedings of the Eighth Italian Conference on Theoretical Computer Science (ICTCS 2003)*, pages 284–301, 2003.

[AGW04] Martin Abadi, Georges Gonthier, and Benjamin Werner. Choice in Dynamic Linking. In *Proceedings of the 7th International Conference FOSSACS 2004 (ETAPS 2004)*, volume 2987 of *LNCS*, pages 12–26, Barcelona, Spain, March 2004. Springer-Verlag.

[AZ04] Davide Ancona and Elena Zucca. Principal Typings for Java-like Languages. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles Of Programming Languages (POPL 2004)*, pages 306–317, Venice, Italy, 2004.

[BD04] Alex Buckley and Sophia Drossopoulou. Flexible Dynamic Linking. In *ECOOP Workshop on Formal Techniques for Java Programs (FTfJP 2004)*, Oslo, Norway, June 2004.

[BMED04] Alex Buckley, Michelle Murray, Susan Eisenbach, and Sophia Drossopoulou. Flexible Dynamic Rotor. Submitted for publication, October 2004.

[DE02] Sophia Drossopoulou and Susan Eisenbach. Manifestations of Dynamic Linking. In *Proceedings of the First Workshop on Unanticipated Software Evolution (USE 2002)*, Malaga, Spain, June 2002. `http://joint.org/use2002/`.

[Dea97] Drew Dean. The Security of Static Typing with Dynamic Linking. In *Proceedings of the Fourth ACM Conference on Computer and Communications Security*, Zurich, Switzerland, April 1997.

[DLE03] Sophia Drossopoulou, Giovanni Lagorio, and Susan Eisenbach. Flexible Models for Dynamic Linking. In Pierpaolo Degano, editor, *Proceedings of the 12th European Symposium on Programming (ESOP 2003)*, volume 2618 of *LNCS*, pages 38–53. Springer-Verlag, April 2003.

[Dod04] Mahesh Dodani. From Objects to Services: A Journey in Search of Component Reuse Nirvana. *Journal of Object Technology*, 3(8):49–54, September 2004. `http://www.jot.fm/issues/issue_2004_08/column5`.

[Dro00] Sophia Drossopoulou. An Abstract Model of Java Dynamic Linking and Loading. In Robert Harper, editor, *Proceedings of the Third International Workshop on Types in Compilation (TIC 2000)*, volume 2071 of *LNCS*, pages 53–84. Springer-Verlag, 2000.

[ECM02] ECMA. *Standard ECMA-335: Common Language Infrastructure.* ECMA International, December 2002. `http://www.ecma-international.org/publications/standards/Ecma-335.htm`.

[EJS02] S. Eisenbach, V. Jurisic, and C. Sadler. Feeling the way through DLL Hell. In *Proceedings of the First Workshop on Unanticipated Software Evolution (USE 2002)*, Malaga, Spain, June 2002. `http://joint.org/use2002/`.

[Fra97] Michael Franz. Dynamic Linking of Software Components. *IEEE Computer*, pages 74–81, March 1997.

[HWC00] Michael Hicks, Stephanie Weirich, and Karl Crary. Safe and Flexible Dynamic Linking of Native Code. In Robert Harper, editor, *Proceedings of the Third International Workshop on Types in Compilation (TIC 2000)*, volume 2071 of *LNCS*, pages 147–176. Springer-Verlag, 2000.

[IPW99] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. In Loren Meissner, editor, *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, pages 132–146, Denver, CO, USA, 1999.

[JLT98] T. Jensen, D. Le Mtayer, and T. Thorn. Security and Dynamic Class Loading in Java: A Formalisation. In *Proceedings of the IEEE International Conference on Computer Languages*, pages 4–15, Chicago, IL, USA, 1998.

[Kre01] Heather Kreger. *Web Services Conceptual Architecture (WSCA 1.0).* IBM, May 2001. `http://www.ibm.com/software/solutions/webservices/pdf/WSCA.pdf`.

[LB98] Sheng Liang and Gilad Bracha. Dynamic Class Loading in the Java Virtual Machine. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'98)*, Vancouver, BC, Canada, October 1998.

[LY99] Tim Lindholm and Frank Yellin. *The Java Virtual Machine.* Addison-Wesley, 1999.

[Mic02] Microsoft. *Shared Source Common Language Infrastructure 1.0 Release*, May 2002. `http://msdn.microsoft.com/net/sscli/`.

[QGC00] Zhenyu Qian, Allen Goldberg, and Alessandro Coglio. A Formal Specification of Java Class Loading. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA 2000)*, pages 325–336, Minneapolis, MN, USA, 2000.

[TBB03] Mark Turner, David Budgen, and Pearl Brereton. Turning Software Into a Service. *IEEE Computer*, pages 38–44, October 2003.

Appendix - Auxiliary definitions

$$M \bullet (\mathtt{m},\mathtt{T}_r,\mathtt{T}_p \mapsto \mathtt{e}) = \begin{cases} (M\backslash(\mathtt{m},\mathtt{T}_r,\mathtt{T}_p \mapsto \_)),(\mathtt{m},\mathtt{T}_r,\mathtt{T}_p \mapsto \mathtt{e}) & \mathit{if}\ (\mathtt{m},\mathtt{T}_r,\mathtt{T}_p) \in \mathit{dom}(M) \\ M,(\mathtt{m},\mathtt{T}_r,\mathtt{T}_p \mapsto \mathtt{e}) & \mathit{otherwise} \end{cases}$$

Fig. 15: Addition of method to $\mathbb{P}_{\mathbb{B}}$ (Virtual table creation)

$$\frac{\mathbb{P}_{\mathbb{C}}(\mathtt{C}) = \mathtt{class\ C\ extends\ C'\ \{\ T\ f;\ \dots\ \}} \qquad \mathit{fields}(\mathbb{P},\mathtt{C'}) = \bar{\mathtt{U}}\ \bar{\mathtt{g}}}{\mathit{fields}(\mathbb{P},\mathtt{C}) = \bar{\mathtt{U}}\ \bar{\mathtt{g}},\mathtt{T}\ \mathtt{f}}$$

$$\frac{\begin{array}{l}\mathbb{P}_{\mathbb{C}}(\mathtt{C}) = \mathtt{class\ C\ extends\ C'\ \{\ C(\bar{T}\ \bar{g},\ U\ f)\ \{\ \dots\ \}\ \}} \\ \mathit{constructor}(\mathbb{P},\mathtt{C'}) = \bar{\mathtt{T}}\end{array}}{\mathit{constructor}(\mathbb{P},\mathtt{C}) = \bar{\mathtt{T}},\mathtt{U}}$$

$$\frac{\begin{array}{l}\mathit{fields}(\mathbb{P},\mathtt{C}_d) = \bar{\mathtt{V}}\ \bar{\mathtt{f}} \\ \mathtt{V}_i = \mathbb{P}_{\mathbb{D}}(\mathtt{T}_f) \qquad \mathtt{f}_i = \mathtt{f} \qquad (\mathtt{V}_j = \mathbb{P}_{\mathbb{D}}(\mathtt{T}_f)\ \wedge\ \mathtt{f}_j = \mathtt{f} \implies j \leq i)\end{array}}{\mathit{offset}(\mathbb{P},\mathtt{f},\mathtt{C}_d,\mathtt{T}_f) = i}$$

$$\frac{\mathbb{P}_{\mathbb{B}}(\mathtt{C}_d)(\mathtt{m},\mathbb{P}_{\mathbb{D}}(\mathtt{T}_r),\mathbb{P}_{\mathbb{D}}(\mathtt{T}_p)) = \mathtt{e}}{\mathit{body}(\mathbb{P},\mathtt{C}_d,\mathtt{m},\mathtt{T}_r,\mathtt{T}_p) = \mathtt{e}}$$

Fig. 16: Functions used at execution

$$\begin{array}{lll}
\sqsubset\cdot\sqsupset^{exe} & ::= & \sqsubset\cdot\sqsupset^{exe}.\mathtt{f}(\!|\_,\_|\!)\ \mid \sqsubset\cdot\sqsupset^{exe}.\mathtt{f}(\!|\_,\_|\!) := \mathtt{e}\ \mid \mathtt{e}.\mathtt{f}(\!|\_,\_|\!) := \sqsubset\cdot\sqsupset^{exe}\ \mid \\
& & \sqsubset\cdot\sqsupset^{exe}.\mathtt{m}(\!|\_,\_,\_|\!)(\mathtt{d})\ \mid \mathtt{e}.\mathtt{m}(\!|\_,\_,\_|\!)(\sqsubset\cdot\sqsupset^{exe})\ \mid \mathtt{y} := \sqsubset\cdot\sqsupset^{exe}\ \mid \\
& & \mathtt{new\ C}(\sqsubset\cdot\sqsupset^{\mathtt{exe}})(\!|\bar{\mathtt{T}}|\!) \\
\\
\sqsubset\cdot\sqsupset^{null} & ::= & \sqsubset\cdot\sqsupset^{null}.\mathtt{f}(\!|\_,\_|\!)\ \mid \sqsubset\cdot\sqsupset^{null}.\mathtt{f}(\!|\_,\_|\!) := \mathtt{e}\ \mid \mathtt{e}.\mathtt{f}(\!|\_,\_|\!) := \sqsubset\cdot\sqsupset^{null}\ \mid \\
& & \sqsubset\cdot\sqsupset^{null}.\mathtt{m}(\!|\_,\_,\_|\!)(\mathtt{d})\ \mid \mathtt{e}.\mathtt{m}(\!|\_,\_,\_|\!)(\sqsubset\cdot\sqsupset^{null})\ \mid \mathtt{y} := \sqsubset\cdot\sqsupset^{null}
\end{array}$$

Fig. 17: Execution contexts