A Featherweight Calculus for Flow-Sensitive Type Systems in Java

David J. Pearce

School of Engineering and Computer Science,
Victoria University of Wellington, New Zealand
djp@ecs.vuw.ac.nz

Abstract

Featherweight Java has been highly successful for reasoning about type systems in Java. However, it is not suited to formalising flow-sensitive type systems. Such systems differ from the norm by allowing variables to have different types at different program points. A large number of problems are naturally expressed in this way. For example, reasoning about non-null types requires retyping a variable `v` after a condition `v!=null`. Other examples include those from data flow analysis, security flow analysis, and more. In this paper, we present Featherweight Intermediate Java (FIJ) — an imperative formalisation of Java. A key advantage of FIJ is support for control-flow arising from exceptions. We formalise the syntax, semantics, and typing process of FIJ. We also discuss what it means for a FIJ program to be type-safe, and detail the proof structure required to show this (which differs considerably from traditional subject-reduction style proofs).

1. Introduction

The complexity of modern programming languages presents a fundamental difficulty for researchers wishing to formalise and reason about language extensions. A good approach here is to develop a "core calculus" of the language which captures features characterising the problem at hand. Featherweight Java (FJ) is a well-known example which provides a core calculus for Java [31]. The aim of FJ was to eliminate as much of the language as possible whilst preserving the key challenges faced in type checking Java. To this end, FJ has been extremely successful, and a host of related systems have arisen. Examples include CJE [3], FIL [33], AliasFJ [2], ConfinedFJ [50], MJ [6], FS [15], FFJ [5], OGJ [45], to name but a few.

Featherweight Java and similar systems have been very successful in helping to formalise type systems as they are traditionally used in programming languages. However, they are rather ineffective at describing algorithms which exploit imperative control-flow, such as those commonly used in program analysis. This is because the vast majority of programming language calculi do not consider control-flow at the type system level. Many do not include imperative control-flow constructs in their syntax (e.g. [31, 3, 15, 23]), whilst those that do normally adopt the standard model where each variable has a single static type (e.g. [16, 6, 45]).

There is a wide range of problems which require a less orthodox approach to typing whereby a variable can have different types at different program points. We refer to this class of problems as the *flow-sensitive typing problems*. A good example here, which arises within the Java language itself, is the problem of *definite assignment* [27]. Here, the aim is to demonstrate that each variable in a Java method is assigned before it can be used. One can think of this as assigning each variable a type from $\{\mathtt{def}, \mathtt{undef}\}$ at each program point. Roughly speaking then, the definite assignment property holds if, for every program point ℓ, those variables used within any expression at ℓ have type `def`. The following illustrates:

```
public void f(int x) {
 int y;
 if(x == 0) {
   y = 0;
 }
 return x + y;
}
```

Hopefully, it is evident that this example is not syntactically correct under definite assignment since `y` may reach the **`return`** statement without being defined. In this case, we would assign `y` the type `undef` after its declaration, `def` after its assignment, and `undef` after the **`if`** statement. Hence, the problem becomes apparent since `y` has type `undef` at the point of use in the **`return`** statement.

There are numerous other properties of interest which can benefit from reasoning about control-flow. Straightforward examples include those from the standard dataflow analysis literature, such as *reaching definitions*, *live variables* and *constant propagation*. More complex, and perhaps more interesting examples include those pertaining to security flow [39], ownership [2], non-null qualifiers [37, 20], read-only qualifiers [47] and more (see e.g. [25, 8, 11, 4, 12, 28, 18]). None of these can be well-described by standard language calculi, such as Featherweight Java. To formalise such systems, the authors either apply techniques from the field of abstract interpretation, or develop ad-hoc flow-sensitive type systems. The difficulty with the former is that it does not provide machinery pertaining to a specific language, whilst those adopting the latter approach typically ignore important control-flow issues. For example, the vast majority of flow-sensitive type systems (e.g. [25, 28, 37]) ignore control-flow arising from exceptions. Likewise, statements which have side-effects are typically forbidden.

This paper presents *Featherweight Intermediate Java (FIJ)*, a calculus suitable for formalising flow-sensitive type systems for Java, such as those discussed above. More specifically, the contributions of this paper are:

1. We present a detailed formalisation of Featherweight Intermediate Java, including operational semantics, typing rules and various correctness proofs.
2. We present a novel approach for encoding control-flow arising from exceptions within an intermediate language. This is advantageous over other approaches, where exceptions are either not considered at all, or handled with separate representations (see e.g. [13, 38, 32]).
3. We present several examples and discuss how they can be formalised by building on Featherweight Intermediate Java.

In the following sections, we will first consider some motivating examples to illustrate the intended use cases for FIJ. Then, we will formalise the syntax of the FIJ language; discuss what it means for a FIJ program to be well-formed and well-typed; and, finally, discuss the problem of typing FIJ programs and provide proofs of soundness and termination.

2. Motivation

Featherweight Intermediate Java (FIJ) is an intermediate representation of Java source code, similar to those found in modern compilers. The purpose of FIJ is to provide a framework within which flow-sensitive type systems can be formalised and proved correct. Such systems assign a static type to each variable, but this can differ between program points.

To achieve this, we have distilled the Java language down to a small core capturing those features relevant to problems in this class. We are particularly concerned with giving a precise treatment of exceptions, since these yield subtle control-paths, but are often overlooked in the presentation of flow-sensitive type systems (for example, [25, 28, 37]).

The main feature of FIJ is that it provides unstructured control-flow, which is ideal for capturing the variety of control structures expressible in Java — particularly those which exploit **break** and **continue** statements. For example, consider the following program:

```
int gcd(int a, int b) {
  while(true) {
    int c = a % b;
    if(c == 0) { break; }
    a = b;
    b = c;
  }
  return b;
}
```

This program can be represented with the following FIJ program:

```
void gcd(int a, int b) {
 var c;
header:
 c = a % b;¹
 if(c == 0) goto exit;²
 a = b;³
 b = c;⁴
 goto header;⁵
exit:
 return b;⁶
}⁷
```

Here, we see that each statement in FIJ is given an integer label ℓ, which is used to uniquely identify it. We also see how **break** statements can be easily encoded as unconditional branches.

In order to simplify the various proofs of interest, FIJ is missing many features from Java. In particular, FIJ does not support: generic types; primitive types other than **boolean** and **int**; arrays; interfaces; and, finally, full constructors. Each of these features could, in principle, be included into FIJ with relative ease. As with Featherweight Java, we would expect people to include the appropriate features for the problem in hand. Thus, FIJ provides a template outlining the main aspects of the proof problem for flow-sensitive type systems, upon which the formalisation of new systems can be built.

Conceptually, FIJ appears similar to other intermediate representations, in particular *Jimple* (part of the SOOT framework [48, 32]) and Java bytecode [36]. While this is superficially true, there are several points to make:

1. The static type of a local variable in FIJ can vary between program points. To determine the static type at a particular point, we must consider the types of values assigned on all control-flow paths reaching it. This is essentially the approach taken with Java bytecode, but contrasts with Jimple where local variables have a single static type for the life of a method.

2. Control-flow arising from explicit or implicit exceptions is formalised within FIJ itself. Again, this contrasts with Jimple where exceptional control-flow is not formalised within the language itself. Instead, reasoning about exceptional control-flow relies on the notion of *exceptional edges* which are made available in SOOT through a separate representation called the *exceptional graph* [32]. This makes it difficult to formally reason about programs in Jimple, not least because the semantics of the exceptional graph are not well defined. Likewise, in Java bytecode, control-flow arising from exceptions is not made explicit at the instruction level and for this reason is often omitted from consideration.
3. FIJ programs are, at the statement and expression level, very similar to Java programs. Thus, programmers can easily see how a particular problem relates to programs at the Java source level. Furthermore, it's easy to see how Java source programs compile down to FIJ (modulo missing features, of course). This helps simplify the formalisation process, since reasoning about control-flow at the Java source level is notoriously difficult. FIJ programs also hides many of the quirks found in other intermediate languages, such as Java bytecode.

Before considering the syntax and semantics of Featherweight Intermediate Java in more detail, we first present some motivating examples to illustrate the kind of problem FIJ is suited to formalising.

2.1 Non-null Types

We consider statically checking Java types annotated with `@NonNull` annotations. Several works address this in detail (see e.g. [20, 4, 14, 10, 17, 37, 30]), and we present a simplified version of the problem here. The following Java method illustrates the main issues:

```
int f(@NonNull String x, String y) {
 int r = 0;

 if(y != null) {
   r += y.size();
 }
 return r + x.size();
}
```

Here, parameter `x` is annotated `@NonNull` to indicate it cannot hold **`null`** on entry to the method and, hence, can be safely dereferenced. However, `y` is not annotated accordingly. Therefore, its type must be updated to `@NonNull` within the conditional body, where this has become known.

We refer to this as a flow-sensitive typing problem since a variable may have different static types at different program points. This problem is particularly suited to formalisation with FIJ, since a more traditional formalisation is cumbersome. For example, the approach taken in JavaCOP [4] is to allow a special cast immediately after a null-check. The following illustrates:

```
int f(@NonNull String x, String y) {
 int r = 0;

 if(y != null) {
  @NonNull String z = (@NonNull String) y;
  r += z.size();
 }
 return r + x.size();
}
```

Here, JavaCOP allows the non-null cast, provided it is the first statement after an appropriate null-check. We consider such code is cumbersome to write and, furthermore, it is difficult (if not impossible) to formalise the correctness of this in a traditional type system. A better solution, we believe, is to formalise the problem as a flow-sensitive type system, for which FIJ is an ideal starting point.

2.2 Instanceof Checking

A related problem to that above, is the issue of dealing with **`instanceof`** tests in Java. For example, consider this common idea:

```
public void f(Object x) {
 if(x instanceof String) {
  String y = (String) x;
  ... // use z as String
 }
}
```

Here, the programmer is forced to statically cast variable `x` to type `String` *even though this cast could never fail*. This seems rather unnecessary, and it would be nice for the compiler to do this automatically. That is, for it to use a flow-sensitive type system whereby variables can be retyped after **`instanceof`** tests. Thus, the above code becomes:

```
void f(Object x) {
 if(x instanceof String) {
  ... // use x as String
 }
}
```

Here, instead of casting `x`, the programmer can simply treat it as having type `String`. Of course, this is only valid within the conditional body. Again, FIJ is ideally suited to formalising a flow-sensitive type system such as this.

2.3 Unique Types

The final example we consider is taken from the AliasJava system [2]. AliasJava supports various type annotations which are related to the problem of object ownership. One of these, the `unique` annotation, indicates a variable has the only reference to a particular object. The following illustrates the main issues:

```
void g(unique Test z) { ... }

unique Test f() {
  unique Test x = new Test();
  unique Test y = new Test();
  g(y);
  y.aMethod();
  return x;
}
```

Here, we see `x` is annotated `unique`, which holds because the object it references is freshly allocated. Since `x` cannot be assigned to any fields in the heap, it is safe for `f`'s return value to be `unique`. In the case of variable `y`, however, we have a problem. This is because it is passed to `g()` as a `unique` reference, but `f()` retains it for the subsequent method call — hence, it is not `unique` in `g()`.

In order to identify the above issue, AliasJava introduces several rules determining how a `unique` variable may be used. One of these states that a variable must be dead once it is read (e.g. as in the invocation `g(y)`). This can be nicely captured using a flow-sensitive type system, whereby the variable `y` is retyped as `dead` after being used. Again, FIJ is ideally suited to formalising such a system. Indeed, by doing this, one can leverage upon FIJ's precise description of control-flow arising from exceptions, leading to a more complete formalisation. In contrast, the formalisation of AliasJava, dubbed AliasFJ, does not consider control-flow arising from exceptions at all.

3. Syntax

The syntax for FIJ is given in Figure 1, where `n` represents the set of variable names, `f` represents the set of field names, `i` the set of integer constants and `b` is the set of boolean constants.

The various constructs representing the different types found in Java are broken into separate categories for: primitives ($\mathtt{T_P}$), class references ($\mathtt{T_C}$), general references ($\mathtt{T_R}$), arbitrary variable types ($\mathtt{T}$) and, finally, method types ($\mathtt{T_F}$). As in Java, not all of these types are expressible at the source level. In particular, the special `null` type cannot be written explicitly in FIJ. We assume that `C` represents the set of fully qualified class names (i.e. including both package and class information). Also, the special *null* type is given to the `null` value and $\top$ is given to variables which hold no value (e.g. they are uninitialised, in dead code, etc). It may seem strange to have the type $\mathtt{T_R}$ for references as, in FIJ, all references are class references (i.e. there are no array references). However, this serves as useful placeholder, which helps to minimise changes when extending FIJ with, for example, arrays. Similarly, the type $\mathtt{T_C}$ is distinct from `C` to simplify the process of adding generic types to FIJ.

The presence of unstructured control-flow constructs (i.e. conditional and unconditional branching) is typical of an intermediate representation, such as Jimple or Java bytecode. These constructs represent a key simplification when describing flow-sensitive typing problems as they reduce the number of constructs to be considered. Another important simplification is that expressions in FIJ are (mostly) side-effect free. In particular, the following Java constructs cannot be represented directly in FIJ and, instead, must be translated to semantically equivalent code:

```
x = ++x + y--;
y = z + (z = 1) + 2;
```

As with Java, we require that every local variable is defined before it is used (known as *definite assignment*).

Another important aspect of FIJ is that field accesses and method invocations are annotated with the static type of the field or method in question. The following illustrates:

```
class Test {
 int field = 0;
 void f(int x) { ... }
 void g() {
  int y = this.[int]field;¹
  f[(int) ⟶ void](this.field);²
  return;³
 }⁴
}
```

This seperates the process of determining what method or field is accessed from the problem of flow-sensitive typing, which is a useful simplification as most problems of interest don't affect method/field lookup. For example, consider non-null example from §2:

```
class Test {
 void f(@NonNull String x) { }
 void g(String x) {
   if(x == null) goto end;¹
   f[(String) ⟶ void](x);²
  end:
   return;³
 }⁴
```

Here, the flow-sensitive typing problem is that of propagating information about which references cannot hold **`null`**. Thus, we would infer that `x!=`**`null`** when `f` is called. Of course, the "nullness type" of `x` has no bearing on which method is actually invoked at statement 2 above — this is fully determined by the standard Java types. Thus, including static type information about method and field accesses simplifies the task of formalising the typing algorithm considerably. Another reason for including static type information about method and field accesses is that it allows us to separate the operational semantics of FIJ programs from the issue of well-typed FIJ programs.

3.1 Constructors

Since the primary purpose of FIJ is to formalise control-flow within methods, we have eliminated constructors altogether.

Types:

$$\begin{array}{lcl}
\mathtt{T_P} & ::= & \mathtt{boolean} \mid \mathtt{int} \\
\mathtt{T_C} & ::= & \mathtt{C} \\
\mathtt{T_R} & ::= & \mathtt{T_C} \\
\mathtt{T} & ::= & \mathtt{T_P} \mid \mathtt{T_R} \mid \mathtt{null} \mid \bot \mid \bot \\
\mathtt{T_F} & ::= & (\overline{\mathtt{T}}) \longrightarrow \mathtt{T}
\end{array}$$

Syntax:

$$\begin{array}{rcl}
\mathtt{D} & ::= & \mathtt{class}\ \mathtt{T_C}\ \mathtt{extends}\ \mathtt{T_C}\ \{\ \overline{\mathtt{T\ f = v;}}\ \overline{\mathtt{M}}\ \} \\
\mathtt{M} & ::= & \mathtt{T\ m(}\overline{\mathtt{T\ x}}\mathtt{)\ throws}\ \overline{\mathtt{T_C}}\ \{\ \overline{\mathtt{var\ v}}\ \ \overline{\mathtt{R}}\ \} \\
\mathtt{R} & ::= & \mathtt{L:} \mid \mathtt{nop;}^{\ell} \mid \mathtt{goto\ L;}^{\ell} \mid \mathtt{S},\ \overline{\mathtt{T_C\ goto\ L}}\mathtt{;}^{\ell} \\
\mathtt{S} & ::= & \mathtt{return\ e} \mid \mathtt{throw\ e} \mid \mathtt{v = e} \mid \mathtt{e.[T]f = e} \mid \mathtt{if(e)\ goto\ L} \\
\mathtt{e} & ::= & \mathtt{e_1\ bop\ e_2} \mid \mathtt{e.[T_F](\overline{e})} \mid \mid \mathtt{(T)\ e} \mid \mathtt{e.[T]f} \mid \mathtt{v} \\
\mathtt{v} & ::= & \mathtt{new\ T_R()} \mid \mathtt{n} \mid \mathtt{i} \mid \mathtt{b} \mid \mathtt{null} \\
\mathtt{bop} & ::= & + \mid - \mid \star \mid / \mid\ < \ \mid\ \leq \ \mid\ \geq \ \mid\ > \ \mid\ == \ \mid !=
\end{array}$$

Figure 1. Syntax for Featherweight Intermediate Java.

Instead, every field is given an initial value. Since there are no constructors which accept parameters, there is no value in considering `super()` calls; likewise, the order in which fields are initialised is no longer of importance, since method calls and field accesses are not permitted. Of course, some systems (e.g. [20]) do need to formalise the construction process, and would need to extend this aspect of FIJ.

3.2 Exceptional Branches

Perhaps the most significant departure from other intermediate representations is the presence of *exceptional branches*. These describe the flow of control that arise when exceptions are thrown. For example, consider the following Java code:

```
int f(int x, int y) {
 try { return x / y; }
 catch(ArithmeticException e) {
   throw e;
}}
```

Exceptional branches allow us to capture the semantics of the **try/catch** block directly within the intermediate representation itself. Thus, the above Java source code is translated into the following FIJ code:

```
int f(int x, int y) {
   var e;
   return x/y, A.Exception goto handler;^1
 handler:
   e = $;^2
   throw e;^3
}^4
```

Here, we abbreviate `ArithmeticException` with the shorthand `A.Exception`. The exceptional branch on statement 1 indicates control should be transferred to `handler` in the event of an `ArithmeticException`. The special variable $ gives access to the exception object's reference.

The semantics of exceptional branches is well-defined. If an exception occurs whilst executing a statement, that statement has no effect and control is transfered along the appropriate exceptional branch (if there is one). Note, we cannot provide exceptional branches to statements *outside of the method in question*; thus, exceptional branches can only represent **try/catch** blocks in the original source code and, if there were none, there will be no exceptional branches. For the purposes of typing FIJ methods, exceptional branches are crucial in assessing the flow of information.

In many cases, it can be statically determined that a particular statement could never take an exceptional branch, and we refer to such edges as *redundant*. Statements in syntactically well-formed FIJ programs may not have exceptional branches which are statically redundant. The rules for determining this are as follows:

- If a statement contains a division expression, then it may have an `ArithmeticException` exceptional branch.
- If a statement contains a field dereference, or method invocation, then it may have a `NullPointerException` exceptional branch.
- If a statement contains a non-primitive cast expression, then it may have a `ClassCastException` exceptional branch.
- If a statement contains a method invocation, then it may have any of the above exceptional branches, as well as any which subtype `RuntimeException`.
- If a statement contains a method invocation for a method declared to throw an exception $\mathtt{T_C}$, then it may have an exceptional branch for type $\mathtt{T_C}$.
- A **throw** statement may have an exceptional branch for any kind of exception.

The purpose of these rules is to conservatively approximate the possible control-flows arising from exceptional edges; as usual, it represents an over-approximation and this can lead to situations where a redundant exceptional branch is not discounted by the above. Consider this FIJ method:

```
int f(int x, int y) {
 var z;
 if(y == 0) goto exit;^1
 return x/y, A.Exception goto handler;^2
exit:
 return 0;^3
handler:
 return z;^4
}^5
```

The reader can easily verify that this method will not divide-by-zero and, hence, raise an `ArithmeticException`.

```
int f(int x, int y) {
 if(x == 0) goto end;^1
  x = x+1;^2
  x = x/y, ArithmeticException goto err;^3
  return x;^4
 end:
  return 0;^5
 err:
  return -1;^6
}^7
```

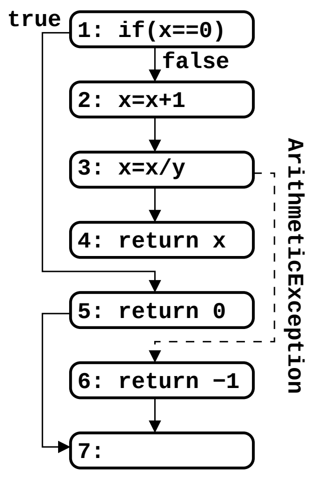

Figure 2. Illustrating a simple FIJ program and its control-flow graph.

However, the method is not well-formed because this cannot be statically verified under the rules given above.

3.3 Well-formedness

For simplicity, we do not provide judgement rules for deciding whether a FIJ class is well-formed. Unfortunately, such rules would be cumbersome to formulate given the nature of FIJ programs. Therefore, we informally consider a FIJ class to be *well-formed* if:

- It is syntactically correct, according to the grammar of Figure 1.
- It does not include any type declarations for `null`, $\bot$ or $\top$ types.
- Every path through a method is terminated by a **`return`** or **`throw`** statement.
- Every variable used in a method is declared in the method's `vars` clause, apart from the special variables `this` and \$.
- No statement in a method has a statically redundant exceptional edge (see above).
- Every branching statement in a method has a corresponding label in that method, and labels are unique in that method.
- Every method is considered to be *well-typed* (see §5).
- Field initialisers are considered to be well-typed under assignment (see §5).

Note, the requirement for definite assignment (i.e. every variable must be defined before use) is implicitly captured in the notion of well-typed methods.

3.4 Control-Flow Graphs

The notion of a control-flow graph is well understood and we now extend this is in a straightforward manner to FIJ programs. Given a FIJ program P, its control-flow graph is a directed graph $G = (V, F, X)$, where V is the set of nodes, F is the set of flow edges, and X is the set of exceptional edges. Every node $v \in V$ corresponds to a unique label in P, with all but one corresponding to a (non-branching) statement. A special exit node, which has no corresponding statement, is used to capture all flow out of the method. Whilst the need for this may seem mysterious, it turns out to be rather useful when typing FIJ programs in §5. The branching statements (i.e. **`if(..) goto`** `L`, **`goto`** `L`) correspond to flow-edges of G. Thus, every flow-edge $f \in F$ is denoted by $v_1 \xrightarrow{\beta} v_2$ which represents an edge from v_1 to v_2 with label $\beta \in \{\mathtt{true}, \mathtt{false}, \bot\}$. For unconditional branches, the special "empty" condition $\bot$ is used. Similarly, every exceptional edge $x \in X$ is denoted by $v_1 \xrightarrow{T_c} v_2$ whose label is the exception being thrown.

Figure 2 illustrates a simple FIJ method and its control-flow graph. There are several important things to note about this example. Firstly, we see that the edges emanating from the **`if`** statement are labelled with **`true`** and **`false`** to indicate which branch is taken when the condition is evaluated. Secondly, the exceptional edges are denoted with dashed arcs, whilst the flow edges are solid. We must distinguish flow edges from exceptional edges because their semantics differs: flow edges are taken *after* the statement in question is executed; exceptional edges are taken *before*. Finally, node 7 represents the special exit which has no statement associated with it. Observe, we do not require a special "entry" node as well, since FIJ programs have only one entry point which is always the the first statement.

3.5 Subtyping

Figure 3 presents the rules for determining whether one type is a subtype of another. In our notation, we take $\mathtt{T}_1 \leq \mathtt{T}_2$ to mean that type $\mathtt{T}_1$ *is a subtype of* $\mathtt{T}_2$. As usual, $\overline{\mathtt{T}}$ is shorthand for a list of types $\mathtt{T}_1, \ldots, \mathtt{T}_\mathtt{n}$. An important property

Subtyping:

$$\frac{\Delta \vdash \mathtt{T}_1 \leq \mathtt{T}_2 \quad \mathtt{T}_2 \leq \mathtt{T}_3}{\Delta \vdash \mathtt{T}_1 \leq \mathtt{T}_3}$$

$$\frac{\Delta \vdash \mathtt{class}\ \mathtt{C}_1\ \mathtt{extends}\ \mathtt{C}_2 \ldots}{\Delta \vdash \mathtt{C}_1 \leq \mathtt{C}_2}$$

$$\frac{}{\Delta \vdash \mathtt{T}_\mathtt{R} \leq \mathtt{java.lang.Object}}$$

$$\frac{}{\Delta \vdash \mathtt{null} \leq \mathtt{T}_\mathtt{R}} \qquad \frac{}{\Delta \vdash \mathtt{T} \leq \top} \qquad \frac{}{\Delta \vdash \bot \leq \mathtt{T}}$$

Figure 3. Subtyping rules for Featherweight Intermediate Java.

of our subtype relation is that it forms a *complete lattice* (i.e. that every pair of types T_1, T_2 has a unique least upper bound, $T_1 \sqcup T_2$, and a unique greatest lower bound, $T_1 \sqcap T_2$). This helps ensure termination of the typing process discussed later in §5. A well-known problem, however, is that Java's subtype relation does not form a complete lattice [34]. This arises because two classes can share the same super-class and implement the same interfaces; thus, they may not have a unique least upper bound. To resolve this, we adopt the standard solution of ignoring interfaces entirely. This works because Java supports only single inheritance between classes.

Finally, we don't consider subtyping between primitives and classes (i.e. boxing/unboxing), since we can assume they have been compiled down to the appropriate conversions at this level.

4. Semantics

Due to the imperative nature of FIJ, we cannot formulate its semantics in the usual manner using reductions. Instead, we adopt a state-machine semantics, where each state is a triple (H, L, pc) consisting of: a heap, H, mapping reference values to object descriptors (see below); a local variable array, L, map variable names to values; and, finally, a program counter pc.

An *object descriptor* is simply a from field names to their values, where the special field $ holds the object's dynamic type. For example, consider the following FIJ class:

```
class Test {
  int x = 0;
  int y = 1;
}
```

Then, the initial object descriptor for an instanceof of `Test` is as follows:

$$\{\$ \mapsto \mathtt{Test}, \mathtt{x} \mapsto 0, \mathtt{y} \mapsto 1\}$$

Whilst this model of Java objects is nice and simple, there are some important differences. In particular, there is no notion of *field hiding* (see [27, 8.3]) where a field in a subclass hides that of a superclass. For example, consider the following Java:

```
class P {
  int x;
}
class C extends P {
  int x;
}
```

Any instance of class `C` will have two fields named `x`, of which one is accessible only within `P`, while the other is accessible only within `C` (and its subclasses). Our notion of object descriptors does not properly support this semantics, although in principle it can be achieved using a pre-processing pass that renames hidden fields as necessary.

Another difference from Java is that FIJ object descriptors lack any notion of the so-called V-table. However, given the object's dynamic type and the method's static type, we can always determine the target of a dynamically dispatched method call.

4.1 Expressions

The (big-step) operational semantics for FIJ expressions are given in Figure 4. We have omitted rules for evaluating multiplication and subtraction, since these are essentially identical to E-B1. Also, the following auxiliary methods are employed:

- **dom**$(\mathtt{H})$. This returns the *domain* of a map. So, for example, **dom**$(\{\mathtt{x} \mapsto 1, \mathtt{y} \mapsto 2\}) = \{\mathtt{x}, \mathtt{y}\}$.
- **field**$(\mathtt{f}, \mathtt{v}, \mathtt{H})$. This looks up the value of the field f in the object referred to by v in heap H. If no such field exists, it simply returns $\bot$.
- **method**$(\mathtt{m}, \mathtt{T}_\mathtt{F}, \mathtt{v}, \mathtt{H})$. This looks up the body of method `m` with type $\mathtt{T}_\mathtt{F}$ in the class of the object referred to by v in heap H, traversing the class hierarchy as necessary. If no such method exists, it simply returns $\bot$.
- **init**$(\mathtt{M}, \overline{\mathtt{e}})$. This creates an initial local variable array for method M which maps its parameters to the supplied expressions, and initialises local variables to $\top$.
- **create**$(\mathtt{T}_\mathtt{R})$. This creates the initial object descriptor for an instance of type $\mathtt{T}_\mathtt{R}$ where all fields are assigned to their initial values.
- **typeof**$(\mathtt{v}, \mathtt{H})$. This returns the type of value v. If v is a reference, then it returns the (dynamic) type of the object referred to by v in heap H; if $\mathtt{v} \notin$ **dom**$(\mathtt{H})$ it simply returns $\bot$.

The rules of Figure 4 are divided into those for propagating values according to the normal course of events, and those for propagating exceptions when they arise. The rules are presented as big-step transitions of the form $(e, L, H_1) \rightsquigarrow (v, H_2)$; this is taken to mean that, given heap H_1 and local

Expression Semantics:

$$\frac{\mathtt{L[v]} \neq \bot}{(\mathtt{v},\mathtt{L},\mathtt{H}) \rightsquigarrow (\mathtt{L[v]},\mathtt{H})}\ \text{[E-V]}$$

$$\frac{\mathtt{i} \in \{\mathtt{null},\mathtt{true},\mathtt{false},\ldots,-1,0,1,2,3,\ldots\}}{(\mathtt{i},\mathtt{L},\mathtt{H}) \rightsquigarrow (\mathtt{i},\mathtt{H})}\ \text{[E-V]}$$

$$\frac{\begin{array}{c}(\mathtt{e}_1,\mathtt{L},\mathtt{H}_1) \rightsquigarrow (\mathtt{v}_1,\mathtt{H}_2) \quad (\mathtt{e}_2,\mathtt{L},\mathtt{H}_2) \rightsquigarrow (\mathtt{v}_2,\mathtt{H}_3)\\ \textbf{typeof}(\mathtt{v}_1,\mathtt{H}_3) = \textbf{typeof}(\mathtt{v}_2,\mathtt{H}_3) = \mathtt{int}\end{array}}{(\mathtt{e}_1 + \mathtt{e}_2,\mathtt{L},\mathtt{H}_1) \rightsquigarrow (\mathtt{v}_1 + \mathtt{v}_2,\mathtt{H}_3)}\ \text{[E-B1]}$$

$$\frac{\begin{array}{c}(\mathtt{e}_1,\mathtt{L},\mathtt{H}_1) \rightsquigarrow (\mathtt{v}_1,\mathtt{H}_2) \quad (\mathtt{e}_2,\mathtt{L},\mathtt{H}_2) \rightsquigarrow (0,\mathtt{H}_3)\\ \textbf{typeof}(\mathtt{v}_1,\mathtt{H}_3) = \textbf{typeof}(\mathtt{v}_2,\mathtt{H}_3) = \mathtt{int}\\ \mathtt{v}_\mathtt{x} \notin \mathtt{H}_3 \quad \mathtt{H}_4 = \mathtt{H}_3[\mathtt{v}_\mathtt{x} \mapsto (\mathtt{A.Exception},\ldots)]\end{array}}{(\mathtt{e}_1\ /\ \mathtt{e}_2,\mathtt{L},\mathtt{H}_1) \rightsquigarrow (\textbf{err}\ \mathtt{v}_\mathtt{x},\mathtt{H}_4)}\ \text{[E-B2]}$$

$$\frac{(\mathtt{e}_1,\mathtt{L},\mathtt{H}_1) \rightsquigarrow (\mathtt{v}_1,\mathtt{H}_2) \quad \textbf{typeof}(\mathtt{v}_1,\mathtt{H}_2) \leq \mathtt{T}_2}{((\mathtt{T}_2)\ \mathtt{e}_1,\mathtt{L},\mathtt{H}_1) \rightsquigarrow (\mathtt{v}_1,\mathtt{H}_2)}\ \text{[E-C1]}$$

$$\frac{\begin{array}{c}(\mathtt{e}_1,\mathtt{L},\mathtt{H}_1) \rightsquigarrow (\mathtt{v}_1,\mathtt{H}_2) \quad \textbf{typeof}(\mathtt{v}_1,\mathtt{H}_2) \not\leq \mathtt{T}_2\\ \mathtt{v}_\mathtt{x} \notin \textbf{dom}(\mathtt{H}_2) \quad \mathtt{H}_3 = \mathtt{H}_2[\mathtt{v}_\mathtt{x} \mapsto (\mathtt{C.C.Exception},\ldots)]\end{array}}{((\mathtt{T}_2)\ \mathtt{e}_1,\mathtt{L},\mathtt{H}_1) \rightsquigarrow (\textbf{err}\ \mathtt{v}_\mathtt{x},\mathtt{H}_3)}\ \text{[E-C2]}$$

$$\frac{\begin{array}{c}(\mathtt{e}_1,\mathtt{L},\mathtt{H}_1) \rightsquigarrow (\mathtt{v}_1,\mathtt{H}_2) \quad \mathtt{v}_1 \neq \mathtt{null}\\ \mathtt{v}_2 = \textbf{field}(\mathtt{f},\mathtt{v}_1,\mathtt{H}_2) \quad \mathtt{v}_2 \neq \bot\end{array}}{(\mathtt{e}_1.\mathtt{f},\mathtt{L},\mathtt{H}_1) \rightsquigarrow (\mathtt{v}_2,\mathtt{H}_2)}\ \text{[E-F1]}$$

$$\frac{\begin{array}{c}(\mathtt{e}_1,\mathtt{L},\mathtt{H}_1) \rightsquigarrow (\mathtt{null},\mathtt{H}_2) \quad \mathtt{v}_\mathtt{x} \notin \textbf{dom}(\mathtt{H}_2)\\ \mathtt{H}_3 = \mathtt{H}_2[\mathtt{v}_\mathtt{x} \mapsto (\mathtt{N.P.Exception},\ldots)]\end{array}}{(\mathtt{e}_1.\mathtt{f},\mathtt{L},\mathtt{H}_1) \rightsquigarrow (\textbf{err}\ \mathtt{v}_\mathtt{x},\mathtt{H}_3)}\ \text{[E-F2]}$$

$$\frac{\begin{array}{c}(\mathtt{e}_1,\mathtt{L},\mathtt{H}_1) \rightsquigarrow (\mathtt{v}_1,\mathtt{H}_2)\ldots(\mathtt{e}_\mathtt{n},\mathtt{L},\mathtt{H}_\mathtt{n}) \rightsquigarrow (\mathtt{v}_\mathtt{n},\mathtt{H}_{\mathtt{n}+1})\\ \mathtt{v}_1 \neq \mathtt{null} \quad \mathtt{M} = \textbf{method}(\mathtt{m},\mathtt{T}_\mathtt{F},\mathtt{v}_1,\mathtt{H}_{\mathtt{n}+1}) \quad \mathtt{M} \neq \bot\\ \mathtt{M} \vdash (\textbf{init}(\mathtt{M},\mathtt{e}_2,\ldots,\mathtt{e}_\mathtt{n}),\mathtt{H}_{\mathtt{n}+1},0) \rightsquigarrow (\mathtt{v}_\mathtt{r},\mathtt{H}_{\mathtt{n}+2})\end{array}}{(\mathtt{e}_1.\mathtt{m}[\mathtt{T}_\mathtt{f}](\mathtt{e}_2,\ldots,\mathtt{e}_\mathtt{n}),\mathtt{L},\mathtt{H}_1) \rightsquigarrow (\mathtt{v}_\mathtt{r},\mathtt{H}_{\mathtt{n}+2})}\ \text{[E-M1]}$$

$$\frac{\begin{array}{c}(\mathtt{e}_1,\mathtt{L},\mathtt{H}_1) \rightsquigarrow (\mathtt{null},\mathtt{H}_2) \quad \mathtt{v}_\mathtt{x} \notin \textbf{dom}(\mathtt{H}_{\mathtt{n}+2})\\ (\mathtt{e}_2,\mathtt{L},\mathtt{H}_2) \rightsquigarrow (\mathtt{v}_2,\mathtt{H}_3)\ldots(\mathtt{e}_\mathtt{n},\mathtt{L},\mathtt{H}_\mathtt{n}) \rightsquigarrow (\mathtt{v}_\mathtt{n},\mathtt{H}_{\mathtt{n}+1})\\ \mathtt{H}_{\mathtt{n}+2} = \mathtt{H}_{\mathtt{n}+1}[\mathtt{v}_\mathtt{x} \mapsto (\mathtt{N.P.Exception},\ldots)]\end{array}}{(\mathtt{e}_1.\mathtt{m}[\mathtt{T}_\mathtt{f}](\mathtt{e}_2,\ldots,\mathtt{e}_\mathtt{n}),\mathtt{L},\mathtt{H}_1) \rightsquigarrow (\textbf{err}\ \mathtt{v}_\mathtt{x},\mathtt{H}_{\mathtt{n}+2})}\ \text{[E-M2]}$$

$$\frac{\begin{array}{c}\mathtt{v}_\mathtt{r} \notin \textbf{dom}(\mathtt{H}_1) \quad \mathtt{M} = \textbf{getconstructor}(\mathtt{m},\mathtt{T}_\mathtt{R})\\ \mathtt{H}_2 = \mathtt{H}_1[\mathtt{v}_\mathtt{r} \mapsto \textbf{create}(\mathtt{T}_\mathtt{R})]\end{array}}{(\mathtt{new}\ \mathtt{T}_\mathtt{r}(),\mathtt{L},\mathtt{H}_1) \rightsquigarrow (\mathtt{v}_\mathtt{r},\mathtt{H}_2)}\ \text{[E-N]}$$

$$\frac{(\mathtt{e}_1,\mathtt{L},\mathtt{H}_1) \rightsquigarrow (\mathtt{v}_1,\mathtt{H}_2)\ldots(\mathtt{e}_\mathtt{k},\mathtt{L},\mathtt{H}_\mathtt{k}) \rightsquigarrow (\textbf{err}\ \mathtt{v}_\mathtt{k},\mathtt{H}_{\mathtt{k}+1}) \quad \mathtt{k} \leq \mathtt{n}}{(\mathtt{e}[\![\mathtt{e}_1,\ldots,\mathtt{e}_\mathtt{n}]\!],\mathtt{L},\mathtt{H}_1) \rightsquigarrow (\textbf{err}\ \mathtt{v}_\mathtt{k},\mathtt{H}_{\mathtt{k}+1})}\ \text{[E-X]}$$

Figure 4. Operational semantics for expressions in Featherweight Intermediate Java. Note, `A.Exception`, `N.P.Exception`, `C.C.Exception` are (respectively) shorthand for `ArithmeticException`, `NullPointerException` and `ClassCastException`.

variable array L, expression e evaluates to value v producing a (potentially) updated heap H_2. Note, we implicitly assume the evaluation relation is transitive. At this point, we see that expressions in FIJ are not entirely side-effect free, since they may contain arbitrary method calls that side-effect the heap.

The semantics of Figure 4 carefully distinguish between *errors* and *exceptions*. Essentially, an error occurs when the machine gets *stuck*; that is, it becomes unable to take any further transitions. On the other hand, an exception arises when it is implicitly, or explicitly thrown. Exceptions are denoted by the special value **err** $\mathtt{v_r}$, where v_r is a reference to the exception object. For example, the following machine state is stuck:

$$(\mathtt{x.f}, \{\mathtt{x} \mapsto \mathtt{v_r}\}, \{\mathtt{v_r} \mapsto \{\$ \mapsto \mathtt{Test}, \mathtt{y} \mapsto 1\}\})$$

Here, we see that local variable `x` maps to reference v_r, which in turn points to an instance of type `Test`. The `Test` class contains a single field `y`, whose value in this case is `1`. Thus, the machine state is stuck because there is no field `f` in class `Test`.

On the other hand, while the following expression is not stuck, it does throw a `NullPointerException`, as indicated by the derivation.

$$\begin{array}{ll}(\mathtt{x.f}, \{\mathtt{x} \mapsto \mathtt{null}\}, \{\}) & \\ \hookrightarrow ((\mathtt{null}).\mathtt{f}, \{\mathtt{x} \mapsto \mathtt{null}\}, \{\}) & \text{(E-V)}\\ \hookrightarrow (\textbf{err}\ \mathtt{v_x}, \{\mathtt{x} \mapsto \mathtt{null}\}, \{\mathtt{v_x} \mapsto \{\$ \mapsto \mathtt{N.P.E}\}\}) & \text{(E-F2)}\end{array}$$

Here, `N.P.E` is shorthand for `NullPointerException`. Now, when an exception is thrown, the key is to ensure the effects of those expressions already evaluated are still visible in the heap. For example, when evaluating a method invocation, the parameters are evaluated left-to-right; then, if an exception is thrown when evaluating one, the effect of those already evaluated must be preserved. This is implemented for all expressions using the special rule E-X. Here, $\mathtt{e}[\![\mathtt{e}_1,\ldots,\mathtt{e}_\mathtt{n}]\!]$ is a shorthand notation for an arbitrary expression whose syntactic form contains expressions $\mathtt{e}_1,\ldots,\mathtt{e}_\mathtt{n}$ in that order of appearance. For example, $\mathtt{e}_1\ \mathtt{bop}\ \mathtt{e}_2$ is matched by $(\bullet\ \mathtt{bop}\ \bullet)[\![\mathtt{e}_1,\mathtt{e}_2]\!]$, where the dots indicate the "holes". Using this shorthand essentially saves us from having to write several near-identical rules.

Statement Semantics:

$$\frac{\mathtt{M(pc) = nop}}{\mathtt{M} \vdash (\mathtt{L, H, pc}) \rightsquigarrow (\mathtt{L, H, pc{+}1})}\;[\text{S-N}] \qquad \frac{\mathtt{M(pc) = goto}\ \ell}{\mathtt{M} \vdash (\mathtt{L, H, pc}) \rightsquigarrow (\mathtt{L, H}, \ell)}\;[\text{S-G}]$$

$$\frac{\begin{array}{c}\mathtt{M(pc) = return\ e}, \overline{\mathtt{T_C\ goto}\ \ell} \\ (\mathtt{e, L, H_1}) \rightsquigarrow (\mathtt{v, L, H_2})\end{array}}{\mathtt{M} \vdash (\mathtt{L, H_1, pc}) \rightsquigarrow (\mathtt{v, H_2})}\;[\text{S-R}]$$

$$\frac{\begin{array}{c}\mathtt{M(pc) = throw\ e}, \overline{\mathtt{T_C\ goto}\ \ell} \\ (\mathtt{e, L, H_1}) \rightsquigarrow (\mathtt{v, L, H_2}) \quad \mathtt{v} \in \mathbf{dom}(\mathtt{H_2}) \\ \mathbf{gethandler}(\mathbf{typeof}(\mathtt{v, L, H_2}), \overline{\mathtt{T_C\ goto}\ \ell}) = \bot\end{array}}{\mathtt{M} \vdash (\mathtt{L, H_1, pc}) \rightsquigarrow (\mathbf{err}\ \mathtt{v, H_2})}\;[\text{S-T1}] \qquad \frac{\begin{array}{c}\mathtt{M(pc) = throw\ e}, \overline{\mathtt{T_C\ goto}\ \ell} \\ (\mathtt{e, L, H_1}) \rightsquigarrow (\mathtt{v, L, H_2}) \quad \mathtt{v} \in \mathbf{dom}(\mathtt{H_2}) \\ \mathbf{gethandler}(\mathbf{typeof}(\mathtt{v, L, H_2}), \overline{\mathtt{T_C\ goto}\ \ell}) = \ell\end{array}}{\mathtt{M} \vdash (\mathtt{L, H, pc}) \rightsquigarrow (\mathtt{L}[\$ \mapsto \mathtt{v}], \mathtt{H_2}, \ell)}\;[\text{S-T2}]$$

$$\frac{\begin{array}{c}\mathtt{M(pc) = n = e}, \overline{\mathtt{T_C\ goto}\ \ell} \\ (\mathtt{e, L, H_1}) \rightsquigarrow (\mathtt{v, L, H_2}) \quad \mathtt{n} \in \mathbf{dom}(\mathtt{L})\end{array}}{\mathtt{M} \vdash (\mathtt{L, H, pc}) \rightsquigarrow (\mathtt{L[n \mapsto v], H_2, pc{+}1})}\;[\text{S-A}] \qquad \frac{\begin{array}{c}\mathtt{M(pc) = e_1.[T]f = e_2}, \overline{\mathtt{T_C\ goto}\ \ell} \\ (\mathtt{e_1, L, H_1}) \rightsquigarrow (\mathtt{v_1, L, H_2}) \quad (\mathtt{e_2, L, H_2}) \rightsquigarrow (\mathtt{v_2, L, H_3}) \\ \mathtt{v_1} \in \mathbf{dom}(\mathtt{H_3}) \quad \mathtt{O_1 = H_3[v_1]} \quad \mathtt{O_2 = O_1[f \mapsto v_2]}\end{array}}{\mathtt{M} \vdash (\mathtt{L, H_1, pc}) \rightsquigarrow (\mathtt{L, H_3[v_1 \mapsto O_2], pc{+}1})}\;[\text{S-F}]$$

$$\frac{\begin{array}{c}\mathtt{M(pc) = if(e_1)\ goto}\ \ell, \overline{\mathtt{T_C\ goto}\ \ell} \\ (\mathtt{e_1, L, H_1}) \rightsquigarrow (\mathtt{false, L, H_2})\end{array}}{\mathtt{M} \vdash (\mathtt{L, H_1, pc}) \rightsquigarrow (\mathtt{L, H_2, pc{+}1})}\;[\text{S-I1}] \qquad \frac{\begin{array}{c}\mathtt{M(pc) = if(e_1)\ goto}\ \ell, \overline{\mathtt{T_C\ goto}\ \ell} \\ (\mathtt{e_1, L, H_1}) \rightsquigarrow (\mathtt{true, L, H_2})\end{array}}{\mathtt{M} \vdash (\mathtt{L, H_1, pc}) \rightsquigarrow (\mathtt{L, H_2}, \ell)}\;[\text{S-I2}]$$

$$\frac{\begin{array}{c}\mathtt{M(pc) = S}\llbracket \mathtt{e_1}, \ldots, \mathtt{e_n} \rrbracket, \overline{\mathtt{T_C\ goto}\ \ell} \\ (\mathtt{e_1, L, H_1}) \rightsquigarrow (\mathtt{v_1, H_2}) \ldots (\mathtt{e_k, L, H_k}) \rightsquigarrow (\mathbf{err}\ \mathtt{v_k, H_{k+1}}) \quad \mathtt{k} \leq \mathtt{n} \\ \mathbf{gethandler}(\mathbf{typeof}(\mathtt{v, H_2}), \overline{\mathtt{T_C\ goto}\ \ell}) = \bot\end{array}}{\mathtt{M} \vdash (\mathtt{L, H_1, pc}) \rightsquigarrow (\mathtt{v, H_2})}\;[\text{S-X1}]$$

$$\frac{\begin{array}{c}\mathtt{M(pc) = S}[[\mathtt{e_1}, \ldots, \mathtt{e_n}]], \overline{\mathtt{T_C\ goto}\ \ell} \\ (\mathtt{e_1, L, H_1}) \rightsquigarrow (\mathtt{v_1, H_2}) \ldots (\mathtt{e_k, L, H_k}) \rightsquigarrow (\mathbf{err}\ \mathtt{v_k, H_{k+1}}) \quad \mathtt{k} \leq \mathtt{n} \\ \mathbf{gethandler}(\mathbf{typeof}(\mathtt{v, H_2}), \overline{\mathtt{T_C\ goto}\ \ell}) = \ell\end{array}}{\mathtt{M} \vdash (\mathtt{L, H_1, pc}) \rightsquigarrow (\mathtt{L}[\$ \mapsto \mathtt{v}], \mathtt{H_2}, \ell)}\;[\text{S-X2}]$$

Figure 5. Operational semantics for statements in Featherweight Intermediate Java

There are a few comments to make about how these rules compare with those of Java. Firstly, the evaluation order follows Java, where subexpressions are evaluated left-to-right (see [27, 15.7]). Secondly, the lack of primitive types such as **`short`** and **`float`** means there is no need to consider binary numeric conversion (see [27, 5.6.2]). Thirdly, Java dictates that abrupt termination of a subexpression (due to a raised exception) results in immediate abrupt termination of the enclosing expression (see [27, 15.6]) — which is also true of FIJ expressions by rule E-X. Finally, division-by-zero, invalid casts and null-dereferences result in the appropriate exception. However, certain expressions in Java can also raise an `OutOfMemoryError` exception (see [27, 15.6]), which we do not consider.

4.2 Statements

The (big-step) operational semantics for FIJ statements are given in Figure 5. The rules are presented as transitions of the form $M \vdash (L_1, H_1, \ell_l) \rightsquigarrow (L_2, H_2, \ell_2)$; this is taken to mean that, given local variable array $\mathtt{L_1}$ and heap $\mathtt{H_1}$, then executing the statement at point ℓ_1 in method M moves the machine into state $(\mathtt{L_2}, \mathtt{H_2}, \ell_2)$. Again, $\mathtt{L_2}$, $\mathtt{H_2}$ and ℓ_2 are the (potentially) updated components, and we implicitly assume the evaluation relation is transitive. Also, $\mathtt{M}(\ell)$ returns the FIJ statement in method $\mathtt{M}$ at point ℓ.

A subtle aspect of our semantics for statements is how return values are handled in rules S-R, S-T1 and S-X1. In this case, the transition rules have the slightly different form $M \vdash (L, H_1, \ell) \rightsquigarrow (v, H_2)$. We can regard this as saying that the method terminates producing value v and (potentially) updated heap H_2. Thus, a requirement that evaluating method M with initial local variable array L and heap H_1 gives result v and updated heap H_2 can be written as follows: $\mathtt{M} \vdash (\mathtt{L}, \mathtt{H_1}, \mathtt{0}) \rightsquigarrow (\mathtt{v}, \mathtt{H_2})$ (recall the evaluation result is transitive). Methods which do not return results (i.e. have **`void`** return type) simply return the value $\bot$.

Again, the semantics of Figure 5 carefully distinguish between errors and exceptions. This time, there are two rules, S-X1 and S-X2, for propagating exceptions. The former represents the case that an exception is thrown which is not

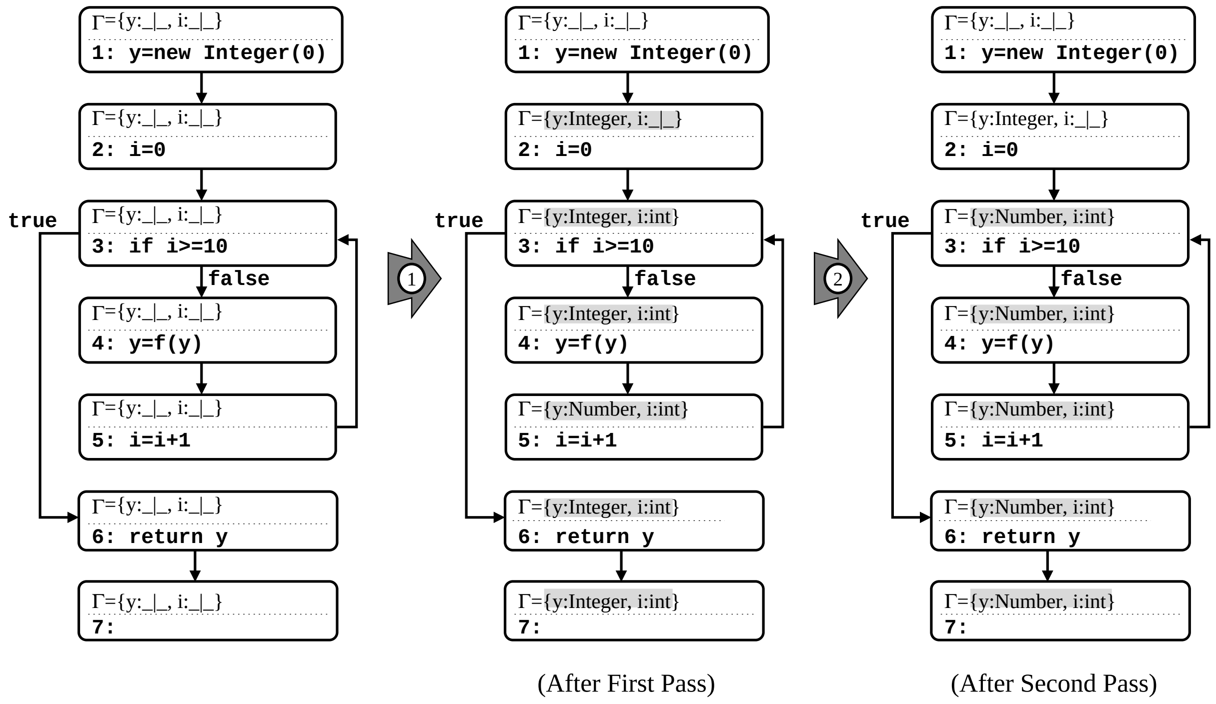

Figure 6. Illustrating our type inference algorithm operating on a simple example. The typing environment immediately before each statement is given. The grey boxes highlight the changes between each iteration.

caught within the method and, hence, propagates out of the method as a return value; the latter represents the case that the exception is caught with control being transferred to its handler. Recall that the special variable \$ is used to store the reference to the exception object, such that it can used in the handler (see §3.2). Also, the method **gethandler**() accepts a type $\mathtt{T_R}$ and a list of exceptional branches $\overline{\mathtt{T_C\ goto}\ \ell}$ and iterates the list, in order of appearance, looking for a type $\mathtt{T_{C1}}$ where $\mathtt{T_r} \leq \mathtt{T_{C1}}$. If no such type is found **gethandler**() returns $\perp$, otherwise it returns the entry point of the handler.

As an example, consider the following snippet of FIJ code:

$$\mathtt{y\ =\ x}^1\mathtt{;}$$
$$\mathtt{x.f\ =\ 1}^2\mathtt{;}$$

Then, executing these two statements from the following state:

$$(\{\mathtt{x} \mapsto \ell_1\}, \{\ell_1 \mapsto \{\mathtt{f} \mapsto \mathtt{0}\}\})$$

yields the following execution trace:

$$\begin{array}{ll} (\{\mathtt{x} \mapsto \ell_1\}, \{\ell_1 \mapsto \{\mathtt{f} \mapsto \mathtt{0}\}\}) & \\ \hookrightarrow (\{\mathtt{x} \mapsto \ell_1, \mathtt{y} \mapsto \ell_1\}, \{\ell_1 \mapsto \{\mathtt{f} \mapsto \mathtt{1}\}\}) & \text{(S-A)} \\ \hookrightarrow (\{\mathtt{x} \mapsto \ell_1, \mathtt{y} \mapsto \ell_1\}, \{\ell_1 \mapsto \{\mathtt{f} \mapsto \mathtt{2}\}\}) & \text{(S-F)} \end{array}$$

At this stage, if there were no more statements in the method, then it would be stuck as no further transitions are possible.

5. Typing

We now wish to consider what it means to type a FIJ program. An important difference between Java and FIJ programs is that variables can have different types at different program points. Therefore, instead of a single typing environment Γ for a given method, we require a separate typing environment for each statement. Then, the typing environment $\Gamma(\ell)$ for some statement S^ℓ captures the types of all variables immediately before that statement is executed. For example, consider the following:

```
int f(int x) {
 var y;
 y = x + 1;¹
 return y;²
}
```

The typing environments for this program are: $\Gamma(1) = \{\mathtt{x} \mapsto \mathtt{int}, \mathtt{y} \mapsto \perp\}, \Gamma(2) = \{\mathtt{x} \mapsto \mathtt{int}, \mathtt{y} \mapsto \mathtt{int}\}$. Observe that, since y is undefined at point 1 it has type $\perp$.

When considering the typing of a FIJ method, the natural question we wish to ask is whether or not a given FIJ method is well-typed. This, in turn, requires asking whether or not the statements and expressions used in that method are well-typed. We now consider each of these points in turn.

5.1 Well-Typedness

To determine whether a FIJ method is well-typed we attempt to construct a valid typing of the method; if this can be constructed, then we say that the method is *well-typed*. However,

Expression Typing:

$$\frac{\mathtt{i} \in \{\ldots,-1,0,1,2,3,\ldots\}}{\Gamma(\ell) \vdash \mathtt{i} : \mathtt{int}}\ \text{[T-CI]} \qquad \frac{\mathtt{b} \in \{\mathtt{false},\mathtt{true}\}}{\Gamma(\ell) \vdash \mathtt{b} : \mathtt{bool}}\ \text{[T-CB]}$$

$$\frac{}{\Gamma(\ell) \vdash \mathtt{null} : \mathtt{null}}\ \text{[T-CN]} \qquad \frac{\{\mathtt{x} \mapsto \mathtt{T}\} \in \Gamma(\ell)}{\Gamma(\ell) \vdash \mathtt{x} : \mathtt{T}}\ \text{[T-V]}$$

$$\frac{\begin{array}{c}\Gamma(\ell) \vdash \mathtt{e}_1 : \mathtt{T}, \mathtt{e}_2 : \mathtt{T} \\ \mathtt{bop} \in \{==, !=, <, <=, >, >=\}\end{array}}{\Gamma(\ell) \vdash \mathtt{e}_1\ \mathtt{bop}\ \mathtt{e}_2 : \mathtt{boolean}}\ \text{[T-B1]} \qquad \frac{\begin{array}{c}\Gamma(\ell) \vdash \mathtt{e}_1 : \mathtt{int}, \mathtt{e}_2 : \mathtt{int} \\ \mathtt{bop} \in \{+,-,/,*\}\end{array}}{\Gamma(\ell) \vdash \mathtt{e}_1\ \mathtt{bop}\ \mathtt{e}_2 : \mathtt{int}}\ \text{[T-B2]}$$

$$\frac{\begin{array}{c}\Gamma(\ell) \vdash \mathtt{e} : \mathtt{T}_1 \\ \mathtt{T}_1 \leq \mathtt{T}_2 \vee \mathtt{T}_2 \leq \mathtt{T}_1\end{array}}{\Gamma(\ell) \vdash (\mathtt{T}_2)\ \mathtt{e} : \mathtt{T}_2}\ \text{[T-C]} \qquad \frac{\begin{array}{c}\Gamma(\ell) \vdash \mathtt{e}_1 : \mathtt{T}_1 \\ \mathbf{isfield}(\mathtt{f}, \mathtt{T}_2, \mathtt{T}_1)\end{array}}{\Gamma(\ell) \vdash \mathtt{e}_0.[\mathtt{T}_2]\mathtt{f} : \mathtt{T}_1}\ \text{[T-F]}$$

$$\frac{\begin{array}{c}\Gamma(\ell) \vdash \mathtt{e} : \mathtt{T} \\ \Gamma(\ell) \vdash \mathtt{e}_1 : \mathtt{P}_1, \ldots, \mathtt{e}_n : \mathtt{P}_n \\ \mathbf{ismethod}(\mathtt{m}, \mathtt{T}_F, \mathtt{T}) \quad \mathtt{T}_F = (\overline{\mathtt{T}}) \rightarrow \mathtt{T}_r\end{array}}{\Gamma(\ell) \vdash \mathtt{e.m}[\mathtt{T}_F](\overline{\mathtt{e}}) : \mathtt{T}_r}\ \text{[T-M]} \qquad \frac{}{\Gamma(\ell) \vdash \mathtt{new\ C()} : \mathtt{C}}\ \text{[T-N]}$$

Figure 7. Typing rules for expressions in Featherweight Intermediate Java.

unlike the traditional approach to type checking, we cannot type the whole method in a single pass; rather, it may require several passes to obtain the correct typing. The following FIJ code covers the main issues:

```
Number f(Number x) { ... }
Number g() {
    var y,i;
    y = new Integer(0);^1
    i = 0;^2
  forheader:
    if(i >= 10) goto forexit;^3
    y = f[(Number) → Number](y)^4
    i = i + 1;^5
    goto forheader;
  forexit:
    return y;^6
}^7
```

Figure 6 illustrates our algorithm typing this method; we use the control-flow graph representation here, since it highlights the possible control-flow paths. Initially, the type of each variable is marked as $\bot$. Then, the algorithm visits every node inferring types based upon the expressions they are assigned (first pass). At this point, the type information is not yet complete and the algorithm must iterate again to propagate the type for y along the back-edge of the loop (thus, reaching a fix-point).

We now consider each aspect of this process in more detail. Then, in §6 we will prove a soundness property of the inferred types; namely, that the type given to every variable is always a supertype of its actual type (at that point).

5.2 Well-Typed Expressions

Inferring the type of an expression in FIJ, given the type environment of its enclosing statement, is essentially identical to that of Java. The typing rules for FIJ expressions are given in Figure 7. These rules are presented using judgements of the form $\Gamma(\ell) \vdash \mathtt{e} : \mathtt{T}$ which are taken to mean that under typing environment $\Gamma(\ell)$, expression e is well-typed and has the type T. The methods **isfield**() and **ismethod**() are predicates which check the determined receiver type does contain an appropriate method/field of the given type. Similarly, **isconstructor**() checks that the class in question has a constructor with the appropriate type.

There are several comments to make about how these rules simplify those of Java. Firstly, given that the lack of primitive types such as **short** and **float**, there is no need to consider binary numeric conversion (see [27, 5.6.2]). Likewise, the lack of interfaces means that T-C can specifically prevent casting an expression which is neither a subtype nor supertype of the target type; in the presence of interfaces, however, this requirement is too strict (see [27, 5.5]). Likewise, the lack of array types prevents any need to consider the unsound approach to subtyping of arrays employed in Java (see [27, 4.10.3]). Finally, the lack of generic types, in particular wildcards and type bounds, means that one does not need to consider capture conversion when typing field, method and constructor accesses (see [27, 5.1.10]).

5.3 Well-Typed Statements

Dealing with typing of statements is somewhat more complex than for expressions. When typing a statement, we need to describe its effect on the typing environment. To do this,

Statement Typing:

$$\frac{\mathtt{M}(\ell) : \mathtt{nop}}{\mathtt{M} \vdash \Gamma(\ell) \longrightarrow \Gamma(\ell)}\ \text{[T-O]}$$

$$\frac{\begin{array}{c}\Gamma(\ell) \vdash \mathtt{e} : \mathtt{T} \\ \mathtt{M}(\ell) : \mathtt{n} = \mathtt{e}, \overline{\mathtt{T_C}\ \mathtt{goto}\ \ell}\end{array}}{\mathtt{M} \vdash \Gamma(\ell) \longrightarrow \Gamma(\ell)[\mathtt{n} \mapsto \mathtt{T}]}\ \text{[T-A1]} \qquad \frac{\begin{array}{c}\Gamma(\ell) \vdash \mathtt{e}_1 : \mathtt{T}_1 \quad \Gamma(\ell) \vdash \mathtt{e}_2 : \mathtt{T}_2 \\ \textbf{isfield}(\mathtt{f}, \mathtt{T}_3, \mathtt{T}_1) \quad \mathtt{T}_2 \leq \mathtt{T}_3 \\ \mathtt{M}(\ell) : \mathtt{e}_1.[\mathtt{T}_3]\mathtt{f} = \mathtt{e}_2, \overline{\mathtt{T_C}\ \mathtt{goto}\ \ell}\end{array}}{\mathtt{M} \vdash \Gamma(\ell) \longrightarrow \Gamma(\ell)}\ \text{[T-A2]}$$

$$\frac{\begin{array}{c}\Gamma(\ell) \vdash \mathtt{e} : \mathtt{T} \quad \mathtt{T} \leq \mathtt{Throwable} \\ \mathtt{M}(\ell) : \mathtt{throw}\ \mathtt{e}, \overline{\mathtt{T_C}\ \mathtt{goto}\ \ell} \\ \textbf{gethandler}(\mathtt{T}, \overline{\mathtt{T_C}\ \mathtt{goto}\ \ell}) = \bot\end{array}}{\mathtt{M} \vdash \Gamma(\ell) \longrightarrow \Gamma(\ell)}\ \text{[T-T]} \qquad \frac{\begin{array}{c}\mathtt{M} : \mathtt{T}_0\ \mathtt{m}(\overline{\mathtt{T}}) \ldots \\ \Gamma(\ell) \vdash \mathtt{e} : \mathtt{T}_1 \quad \mathtt{T}_1 \leq \mathtt{T}_0 \\ \mathtt{M}(\ell) : \mathtt{return}\ \mathtt{e}, \overline{\mathtt{T_C}\ \mathtt{goto}\ \ell}\end{array}}{\mathtt{M} \vdash \Gamma(\ell) \longrightarrow \Gamma(\ell)}\ \text{[T-R]}$$

$$\frac{\begin{array}{c}\Gamma(\ell) \vdash \mathtt{e} : \mathtt{boolean} \\ \mathtt{M}(\ell) : \mathtt{if(e)}\ \mathtt{goto}\ \ell_\mathtt{T}, \overline{\mathtt{T_C}\ \mathtt{goto}\ \ell}\end{array}}{\mathtt{M} \vdash \Gamma(\ell) \xrightarrow{\mathtt{true}} \Gamma(\ell)}\ \text{[T-I1]} \qquad \frac{\begin{array}{c}\Gamma(\ell) \vdash \mathtt{e} : \mathtt{boolean} \\ \mathtt{M}(\ell) : \mathtt{if(e)}\ \mathtt{goto}\ \ell_\mathtt{T}, \overline{\mathtt{T_C}\ \mathtt{goto}\ \ell}\end{array}}{\mathtt{M} \vdash \Gamma(\ell) \xrightarrow{\mathtt{false}} \Gamma(\ell)}\ \text{[T-I2]}$$

Figure 8. Typing rules for statements in Featherweight Intermediate Java.

we use the standard notion of a *transfer function*. For a given method and statement, this accepts the incoming typing environment and produces an updated typing environment capturing the statement's effect.

The transfer function is denoted by $\mathtt{f_M}(\Gamma(\ell), \beta)$ where ℓ identifies the statement in question, $\Gamma(\ell)$ is the typing environment immediately before that statement, and β is the outgoing edge label (one of `true`, `false` or $\bot$). To understand this more clearly, consider the simplest example:

```
void f(String x) { nop^1 }^2
```

The typing environment immediately before the `nop` statement will be $\Gamma(1) = \{\mathtt{x} \mapsto \mathtt{String}\}$. To determine the typing environment which holds after the `nop` statement (i.e. $\Gamma(2)$), we apply the transfer function to $\Gamma(1)$; that is, we compute $\mathtt{f_M}(\Gamma(1), \bot)$, which gives $\Gamma(2) = \{\mathtt{x} \mapsto \mathtt{String}\}$. Thus, the transfer function has had no effect on the incoming typing environment `Gamma(1)` which reflects the operational semantics of `nop`. In this case, we supplied $\bot$ for the outgoing edge label β, since the statement in question was not a conditional branch.

To see where the outgoing edge label becomes useful, we must consider a more complex example:

```
void f(String x) {
  if(x == null) goto fbranch^1;
  nop^2
  return;^3
 fbranch:
  nop^4
  return;^5
}^4
```

Here, we again have $\Gamma(1) = \{\mathtt{x} \mapsto \mathtt{String}\}$. Now, consider computing $\Gamma(2)$ versus computing $\Gamma(4)$. In some flow-sensitive type systems (e.g. the non-null system from §2) we want to observe a difference between $\Gamma(2)$ and $\Gamma(4)$. For example, that $\Gamma(2) = \{\mathtt{x} \mapsto \mathtt{@NonNull\ String}\}$ whilst $\Gamma(4) = \{\mathtt{x} \mapsto \mathtt{null}\}$. Therefore, we need the transfer function to behave differently, depending upon whether its result will propagate across a `true` or `false` edge. Thus, the transfer function is parameterised on the outgoing edge label to permit this.

Figure 8 gives the rules which define the transfer function $\mathtt{f_M}(\Gamma(\ell), \beta)$ for a given method `M`. These are presented using judgements of the form $\mathtt{M} \vdash \Gamma(\ell) \xrightarrow{\beta} \Gamma$. These are taken to mean that, for method `M`, we have $\mathtt{f_M}(\Gamma(\ell), \beta) = \Gamma$, where Γ is the updated version of $\Gamma(\ell)$. Aside from determining an updated typing environment, the transfer function also encodes various constraints that must hold for the statement to be considered type safe. For example, in T-A2 that the type of an expression is a subtype of the field it is assigned to. Also, it may seem strange that the transfer function does not generally produce an updated typing environment. For example, the type of fields which are assigned is not updated. This reflects the fact that we cannot safely retype fields in Java because of concurrency[1]. Similarly, the transfer function does not produce different environments on either side of a conditional. This reflects the fact that our base system does not stand to gain from doing this. However, by providing the two rules T-I1 and T-I2, we are effectively leaving a place holder for systems extending our calculus that can benefit from this.

[1] Although, if a method is marked as **`synchronized`** this could be done safely and systems based on our calculus may choose to do this.

5.4 Well-Typed Methods

To determine whether a method is well-typed or not, we must computing a valid typing of the entire method — it is insufficient to look at each statement in isolation. Thus far, we have described how the typing of statements and expressions produces a typing environment which holds after the statement, in terms of that which held before. However, it remains to connect these input and output stores together, according to the control-flow of the method in question. In particular, at join-points in the control-flow graph (i.e. nodes with more than one predecessor) we must combine stores from all incoming paths.

To define how stores are combined at join-points in the control-flow graph, we must introduce a partial ordering on typing environment. Thus, $\Gamma(\ell_1) \sqsubseteq \Gamma(\ell_2)$ implies that $\Gamma(\ell_1)$ must be a lower bound of $\Gamma(\ell_2)$.

DEFINITION 1. *Let* $\Gamma(\ell_1)$ *and* $\Gamma(\ell_2)$ *be typing environments. Then,* $\Gamma(\ell_1) \sqsubseteq \Gamma(\ell_2)$ *holds iff* $\textbf{\textit{dom}}(\Gamma(\ell_1)) = \textbf{\textit{dom}}(\Gamma(\ell_2))$ *and* $\forall \mathtt{v} \in \textbf{\textit{dom}}(\Gamma(\ell_1)).\big[\Gamma_1[\mathtt{v}] \leq \Gamma(\ell_2)[\mathtt{v}]\big]$.

So, for example, the following orderings are trivially valid:

$$\{\mathtt{v} \mapsto \mathtt{String}\} \sqsubseteq \{\mathtt{v} \mapsto \mathtt{Object}\}$$
$$\{\mathtt{v} \mapsto \mathtt{String}\} \sqsubseteq \{\mathtt{v} \mapsto \bot\}$$

On the otherhand, the following do not hold:

$$\{\mathtt{v} \mapsto \mathtt{String}\} \not\sqsubseteq \{\mathtt{v} \mapsto \mathtt{int}\}$$
$$\{\mathtt{v} \mapsto \bot\} \not\sqsubseteq \{\mathtt{v} \mapsto \mathtt{String}\}$$

The purpose of the partial ordering is to capture compatible "type flows", whilst preventing incompatible ones. Consider the following FIJ snippet:

```
 x = new Integer(...);¹
 if(...) goto exit;²
 x = new Float(...);³
exit:
 nop;⁴
```

Here, the question is, what type does x have in $\Gamma(4)$? We know that x references either an `Integer` or `Float` object, and thus we desire the *most precise type* capturing this. Therefore, we have the following ordering requirements between environments:

$$\Gamma(1)[\mathtt{x} \mapsto \mathtt{Integer}] \sqsubseteq \Gamma(2)$$
$$\Gamma(2) \sqsubseteq \Gamma(3)$$
$$\Gamma(2) \sqsubseteq \Gamma(4)$$
$$\Gamma(3)[\mathtt{x} \mapsto \mathtt{Float}] \sqsubseteq \Gamma(4)$$

Since the environment ordering is transitive, these reduce to the following requirements on $\Gamma(4)$:

$$\Gamma(1)[\mathtt{x} \mapsto \mathtt{Integer}] \sqsubseteq \Gamma(4)$$
$$\Gamma(3)[\mathtt{x} \mapsto \mathtt{Float}] \sqsubseteq \Gamma(4)$$

Thus, any valid typing for this method must ensure that $\mathtt{Integer} \leq \Gamma(4)[\mathtt{x}]$ and $\mathtt{Float} \leq \Gamma(4)[\mathtt{x}]$ hold. Then, the most precision solution for x corresponds to the least upper bound of `Integer` and `Float` (i.e. `Number`).

To formulate the ordering requirements precisely, we use *typing equations* which are defined as follows:

DEFINITION 2. *Let* $G = (V, F, X)$ *be the control-flow graph for a method* M. *Then, the typing equations are given by* $\Gamma(\mathtt{y}) = \Big(\bigsqcup_{\mathtt{x} \xrightarrow{\beta} \mathtt{y} \in \mathtt{F}} \mathtt{f_M}(\Gamma(\mathtt{x}), \beta)\Big) \sqcup \Big(\bigsqcup_{\mathtt{x} \xrightarrow{\mathtt{T_c}} \mathtt{y} \in \mathtt{X}} \Gamma(\mathtt{x})[\$ \mapsto \mathtt{T_c}]\Big)$.

Here, $\mathtt{f_M}(\Gamma(\ell), \beta)$ is the transfer function discussed previously. The formula for the typing equations has two components: the left-side captures the normal execution flow, and is defined over the flow edges F; whilst, the right-side captures exceptional execution flow, and is defined in terms of the exceptional edges X.

Understanding the typing equations is best achieved through an example. Consider the following simple FIJ method:

```
int f(String x) {
 var y;
 y = x.size[void → int](),
     NullPointerException goto handler;¹
 return y;²
handler:
 return y;³
}⁴
```

Firstly, it should be clear that this method is not well-typed since y is not defined at `handler` and, hence, we should identify this during typing. Secondly, `N.P.E` is shorthand for `NullPointerException`. Then, the typing equations for this method are:

$$\begin{aligned}
\Gamma(1) &= \{\mathtt{x} \mapsto \mathtt{String}, \mathtt{y} \mapsto \top\}\\
\Gamma(2) &= \mathtt{f_M}(1, \Gamma(1), \bot)\\
\Gamma(3) &= \Gamma(1)[\$ \mapsto \mathtt{NullPointerException}]\\
\Gamma(4) &= \mathtt{f_M}(2, \Gamma(2), \bot) \sqcup \mathtt{f_M}(\Gamma(3), \bot)
\end{aligned}$$

Following the rules of Figure 8 for statement typing and Definition 1 for environment ordering , the above equations expand as follows:

$$\begin{aligned}
\Gamma(1) &= \{\mathtt{x} \mapsto \mathtt{String}, \mathtt{y} \mapsto \top\}\\[1em]
\Gamma(2) &= \mathtt{f_M}(\Gamma(1), \bot)\\
&\quad \hookrightarrow \Gamma(1)[\mathtt{y} \mapsto \mathtt{int}]\\
&\quad \hookrightarrow \{\mathtt{x} \mapsto \mathtt{String}, \mathtt{y} \mapsto \mathtt{int}\}\\[1em]
\Gamma(3) &= \Gamma(1)[\$ \mapsto \mathtt{NullPointerException}]\\
&\quad \hookrightarrow [\mathtt{x} \mapsto \mathtt{String}, \mathtt{y} \mapsto \top, \$ \mapsto \mathrm{N.P.E}]\\[1em]
\Gamma(4) &= \mathtt{f_M}(\Gamma(2), \bot) \sqcup \mathtt{f_M}(\Gamma(3), \bot)\\
&\quad \hookrightarrow \{\mathtt{x} \mapsto \mathtt{String}, \mathtt{y} \mapsto \mathtt{int}\} \sqcup ?
\end{aligned}$$

Here, the fact that the method is not type safe manifests itself, since we cannot evaluate $\mathtt{f_M}(\Gamma(3), \bot)$. This is because, the type rule T-R requires returned expression's type be a subtype of the method's return type. Since $\mathtt{y} \mapsto \top \in \Gamma(3)$, this translates into a requirement that $\top \leq \mathtt{int}$ which does not hold.

Finally, the typing equations can be solved in the usual manner by iterating until a fixed-point is reached using a *worklist algorithm*. Although beyond the scope of our consideration here, there are numerous strategies which can be employed to make this efficient [29, 7, 21, 22, 19, 41]. However, in the following section, we will be concerned with *showing that solving these equations is possible* — that is, that they will eventually reach a fixed-point.

6. Soundness

We now demonstrate that our flow-sensitive typing system *terminates* and is *correct*. The traditional progress and preservation proofs used in FJ style calculus, however, cannot be applied here. Instead, we must adopt a proof structure from the abstract interpretation and dataflow analysis communities. The idea is to provide a proof structure which can be reused when showing correctness of a particular system based on FIJ.

6.1 Termination

Demonstrating termination amounts to showing that the typing equations of Definition 2 always have a *least fixed-point*. In particular, this requires showing that our ordering relation on typing environments is a *join-semilattice* (i.e. that any two typing environments always have a unique least upper bound) and that the transfer function, $\mathtt{f_M}()$, is monotonic. These are addressed by Lemmas 1 and 2.

Showing that the ordering relation on typing environments is a *join-semilattice* is fairly easy in the default case. This is because the subtyping relation forms a complete lattics (recall §3). However, in systems where this is not so (for example, where the full class hierarchy including interfaces is considered), then this proof can be significantly more challenging.

LEMMA 1. *Let Γ_1 and Γ_2 be typing environments, such that* $\textbf{\textit{dom}}(\Gamma_1) = \textbf{\textit{dom}}(\Gamma_2)$. *Then, $\Gamma_1 \sqcup \Gamma_2$ is a typing environment where $\{\Gamma_1, \Gamma_2\} \sqsubseteq \Gamma_1 \sqcup \Gamma_2$.*

PROOF 1. *Suppose not. Then, we have two Γ_1 and Γ_2 where $\textbf{\textit{dom}}(\Gamma_1) = \textbf{\textit{dom}}(\Gamma_2)$ and either $\Gamma_1 \sqcup \Gamma_2$ does not exist, or one of $\Gamma_1 \not\sqsubseteq \Gamma_1 \sqcup \Gamma_2$ and $\Gamma_2 \not\sqsubseteq \Gamma_1 \sqcup \Gamma_2$ holds. Since the subtype relation is a lattice, $\Gamma_1 \sqcup \Gamma_2$ must be a typing environment. W.L.O.G. assume $\Gamma_1 \not\sqsubseteq \Gamma_1 \sqcup \Gamma_2$. Again this is a contradiction since, by definition, we know that $\forall v \in \textbf{\textit{dom}}(\Gamma_1).\big[(\Gamma_1 \sqcup \Gamma_2)[\mathtt{v}] = \Gamma_1[\mathtt{v}] \sqcup \Gamma_2[\mathtt{v}]\big]$.*

Showing that the transfer function $\mathtt{f_M}()$ is monotonic forms a crucial part of the inductive proof that the typing equations can always be solved. To do this, we need to show that if the following holds:

$$\mathtt{f_M}(\Gamma_1(\ell), \beta) \rightarrow \Gamma_1$$
$$\mathtt{f_M}(\Gamma_2(\ell), \beta) \rightarrow \Gamma_2$$

where $\Gamma_1(\ell) \sqsubseteq \Gamma_2(\ell)$, then it is always the case that $\Gamma_1 \sqsubseteq \Gamma_2$. Intuitively, one can think of this as saying the following: suppose at some point whilst solving the typing equations we have a typing environment for some program point ℓ; to ensure termination we need that, in subsequent iterations, the typing environment at that point *only gets greater*. If this is so, then either the typing environment at that point reaches a fixed-point, or it reaches $\top$ — whichever happens first.

LEMMA 2. *The typing equations from Definition 2 are monotonic.*

PROOF 2. *Assume for some point ℓ we have $\Gamma_1(\ell), \Gamma_2(\ell)$ where $\Gamma_1(\ell) \sqsubseteq \Gamma_2(\ell)$, and also that $\mathtt{f_M}(\Gamma_1(\ell), \beta) \rightarrow \Gamma_1$ and $\mathtt{f_M}(\Gamma_2(\ell), \beta) \rightarrow \Gamma_2$. Then, we need to show that $\Gamma_1 \sqsubseteq \Gamma_2$. We proceed by case analysis on the different kinds of statement to show this always holds. Considering Figure 8, the only rule which actually updates $\Gamma(\ell)$ is T-A1. For this case, it follows that $\Gamma_1 = \Gamma_1(\ell)[n \mapsto T_1]$ and $\Gamma_2 = \Gamma_2(\ell)[n \mapsto T_2]$. Thus, we need only to show that $\mathtt{T_1} \leq \mathtt{T_2}$, which is done by case analysis on the typing rules for expressions (Figure 7). In particular, since the types of fields and methods are fixed, only variable reads (i.e. rule T-V) could introduce a problem. Consider an expression $\mathtt{e}$ involving variables $\overline{\mathtt{v}}$. Since $\Gamma_1(\ell) \sqsubseteq \Gamma_2(\ell)$ it follows that for each $v \in \overline{v}$, we have $\Gamma_1(\ell)[\mathtt{v}] \leq \Gamma_2(\ell)[\mathtt{v}]$. By inspection of the typing rules for expressions, it becomes apparent that in such case $\mathtt{e}$ has the same type under $\Gamma_1(\ell)$ and $\Gamma_2(\ell)$. Note that, it may also occur that $\mathtt{e}$ does not have a valid type under $\Gamma_1(\ell)$ or $\Gamma_2(\ell)$ (e.g. because some field is not present as required). Such a situation indicates a syntactic error in the FIJ method.*

The case analysis for the above proof of Lemma 2 was made simple because only rule, T-A1, actually updated the typing environment. In more complex systems, such as those which account for the effects of conditions, we expect one must consider more rules in the case analysis.

6.2 Correctness

The aim now is to show that the typing equations produce a typing which is correct with respect to the *typing property*; namely, that a local variable given type `T` at point ℓ can only ever hold values in `T`. For example, a variable given type `int` can only hold integers; or, in a non-null system, that a variable given type `@NonNull T` can never hold `null`. The key is that we have to define a relationship between the types and the runtime values of the system (typically referred to as the *concretization* and *abstraction functions*). In our case, this connection is fairly self evident from the simple values we have (i.e. integers, booleans, references and exceptions) and so there is no need to do this formally. For more interesting systems which build upon FIJ, the link between types and values may not always be so apparent, and a formal connection will be required.

LEMMA 3. *Let Γ be a valid typing for some method* `M`*, and* `v` *some variable at an arbitrary point ℓ in* `M`*. Suppose also that*

$\Gamma(\ell)[\mathtt{v}] = \mathtt{T}$. *Then, in no execution of* M *can* v *hold a value which is not in* T.

PROOF 3. *Suppose not. Then, there exists some execution path through the method where, at some point ℓ, a variable v has type* T *but its runtime value is not in* T. *W.L.O.G. let ℓ be the first such point in the execution path. Then, the type of* v *coming from some predecessor of ℓ is incorrect. We now traverse backwards from each predecessor of ℓ looking for the first point ℓ' where v has type* $\mathtt{T}'$ *and its runtime value is in* $\mathtt{T}'$ *(i.e.* v *is typed correctly). In some cases (e.g. a loop), we may reach ℓ again, in which case, we can discount that avenue. However, there must be at least one such point ℓ'. Finally, we eliminate this by a case analysis on the different statement types in roughly the same manner as for Lemma 2.*

A key aspect of the proof for Lemma 3 is a backwards traversal to find the offending transfer function. For this to work correctly, we require that for every possible state transition $(L_1, H_2, \ell_1) \longrightarrow (L_2, H_2, \ell_2)$ there is a corresponding edge in the control-flow graph. This is easy to verify, since every sequential and branching statement corresponds to a flow edge, and every exceptional branch corresponds (conservatively) to an exceptional edge.

7. Related Work

Since the pioneering work of Drossoupolu and Eisenbach [16], the study of Java has become increasingly sophisticated and precise. Featherweight Java (FJ) has been perhaps the most successful vehicle in this regard [31]. The idea behind FJ was to reduce the language as much as possible, whilst retaining a core system whose type-soundness proof was non-trivial, and captured the main issues faced in type checking Java. The resulting system was purely functional, and had no true notion of the program heap. Numerous works have built upon FJ, either as a starting for proving novel language extensions (e.g. [2, 45, 50, 5]), or to model some feature more precisely [3, 6, 9].

A common difficulty arises amongst designers of Java calculi when there is a need to model control-flow. Typically, this arises because the problem being formalised necessitates a mutable heap model — ownership systems are perhaps a good example here (e.g. [2, 45]). The issue is that Java permits a wide range of control-flow constructs, with those arising from break/continue statements, labelled blocks and exceptions being particularly challenging. To deal with this, most calculi (e.g. [2, 6, 45]) adopt a simple model of control-flow, excluding looping constructs, break/-continue statements, labelled blocks and try-catch handlers. Of course, this is acknowledged and often seems a necessary evil to achieve any degree of simplicity and elegance. We argue that, by formalising such systems at the intermediate language level (as in FIJ), rather than the source level, one can model such constructs with relative ease. Of course, the disadvantage is that the formalism does not apply directly to the source language. However, we believe that FIJ does a good job of finessing this issue, since it is easy for the reader to see how source constructs translate into FIJ.

7.1 Flow-Sensitive Typing Problems

Perhaps the most relevant work to FIJ, is that of CQual [24, 25]. CQual is a flow-sensitive qualifier inference algorithm which supports numerous type qualifiers, including those related to synchronisation and file I/O. CQual was formalised using an extension of the lambda calculus which included imperative assignment, but no other forms of control-flow. Furthermore, their system differs from FIJ, in that no effort was made to account for the effects arising from conditionals. Building on this is the work of Chin *et al.* which also supports numerous qualifiers, including `nonzero`, `unique` and `nonnull` [11, 12]. Again, their system was formalised using an extension of the lambda calculus supporting assignment, but no other forms of control-flow. Likewise, the effect of conditionals could not be accounted for, which severely restricted their use of `nonnull` qualifiers. JQual extended these systems to Java, and considered whole-program inference [28]. However, JQual does not perform flow-sensitive inference, and its formalisation reflects this. An interesting question is how JQual deals with control-flow arising from exceptions, but this is not discussed at all.

AliasJava introduces several qualifiers for reasoning about object ownership [2]. The `unique` qualifier indicates a variable holds the only reference to an object; similarly, `owned` is used to confine an object to the scope of its enclosing "owner" object. Formalising the correctness of this system requires reasoning about control-flow; in particular, as discussed in §2, one must show that a variable marked `unique` is dead after it has been used. However, the formalisation of AliasJava avoids this by ignoring all control-flow except field assignment. Instead, the authors comment that the full implementation exploits a live variable analysis to achieve the necessary correctness checks.

Ekman *et al.* implemented a non-null checker within the JustAdd compiler [17]. While their system is flow-sensitive and accounts for the effect of conditionals, they did not formalise it. Pominville *et al.* also discuss a flow-sensitive non-null analysis built using SOOT that accounts for conditionals, but again was not formalised [44]. JavaCOP provides an expressive language for writing type system extensions, such as non-null types [4]. This system cannot account for the effects of conditionals; however, as a work around, the tool allows assignment from a nullable variable `x` to a non-null variable if this is the first statement after a `x!=null` conditional. The JACK tool for bytecode verification of `@NonNull` types has some relation to our approach [37]. This extends the bytecode verifier with an extra level of indirection called *type aliasing*. This technique tracks local and stack locations which are known to aliases, thus enabling the verification of `@NonNull` types. In particular, this enables the system to retype a variable `x` as `@NonNull` in the body a **if**(x!=**null**) conditional. The algorithm is formalised using a flow-sensitive type system operating on Java bytecode. Finally, it is worth noting that none of these works on non-null types discuss how control-flow arising from exceptions is handled.

The work of Gagnon *et al.* presents a technique for converting Java Bytecode into an intermediate representation [26]. Key to this is the ability to infer static types for the locals and stack locations used in the bytecode. Their problem is rather different from ours, since it is about inferring a single static type for each variable. Since variables are untyped in Java bytecode, this is not always possible as a variable can — and often does — have different types at different points; in such situations, they split variables as necessary into two or more variables, with each having a different static type. The system was not formalised, although the implementation used SOOT and, hence, presumably considered control-flow arising from exceptions.

7.2 Type Inference

Related work also exists on type inference for Object-Oriented languages (e.g. [40, 43, 49]). These, almost exclusively, assume the original program is completely untyped and employ set constraints (see [1]) for inferring types. This proceeds across method calls, necessitating knowledge of the program's call graph (which must be approximated in languages with dynamic dispatch). Typically, a constraint graph representing the entire program is held in memory at once, making these approaches somewhat unsuited to separate compilation [40]. Such systems share a strong relationship with other constraint-based program analyses, such as *points-to* analysis (e.g. [35, 46, 41, 42]). Typically, these systems are not formalised in terms of the target language and, hence, formalisation with FIJ would be of interest here. One issue, however, is that FIJ essentially considers only intra-procedural typing problems, whilst these are primarily interprocedural problems. Nevertheless, FIJ could be extended to support this, and in doing so would provide a more complete description of these systems.

7.3 Exceptions

The treatment of exceptions in intermediate representations has received relatively scant attention. The standard approach is to employ some form of CFG representation extended with exceptional edges (see e.g. [13, 38, 32]). As in FIJ, the exceptional edges capture the flow of control which arises when an exception is raised. However, are not aware of any attempts to formalise system which are based on these representations. This is one of key contributions of FIJ, since by incorporating exceptional control-flow into the intermediate representation directly, it becomes more amenable to formalisation.

8. Conclusion

We have presented Featherweight Intermediate Java (FIJ), a language ideally suited to formalising and reasoning about flow-sensitive typing systems. A key aspect of FIJ is that control-flow arising from exceptions is modelling within the language itself, rather than relying on a separate representation. In this way, it becomes relatively easy for the formalisations of flow-sensitive type systems to how exceptional flow is handled. Hopefully, this will lead to more complete formalisations of such systems, whose current formalisations typically ignore exceptional flow. We have presented several motivating flow-sensitive typing problems, provided an operational semantics, typing rules and proof structure for FIJ. In the future, we believe it would be interesting to extend FIJ to support interprocedural typing problems, as this would increase the range of problems it could be applied to.

Acknowledgements. Thanks to Prof. James Noble and Sriram Srinivasan for helpful comments on earlier drafts of this paper.

References

[1] A. Aiken. Introduction to set constraint-based program analysis. *Science of Computer Programming*, 35(2–3):79–111, 1999.

[2] J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *Proc. OOPSLA*, pages 311–330, 2002.

[3] D. Ancona, G. Lagorio, and E. Zucca. A core calculus for java exceptions. In *Proc. OOPSLA*, pages 16–30, 2001.

[4] C. Andreae, J. Noble, S. Markstrum, and T. Millstein. A framework for implementing pluggable type systems. In *Proc. OOPSLA*, pages 57–74. ACM Press, 2006.

[5] S. Apel, C. Kästner, and C. Lengauer. Feature featherweight java: a calculus for feature-oriented programming and stepwise refinement. In *Generative Programming and Component Engineering GPCE*, pages 101–112. ACM, 2008.

[6] G. Bierman, M. Parkinson, and A. Pitts. MJ: An imperative core calculus for Java and Java with effects. Technical Report 563, Cambridge University Computer Laboratory, Apr. 2003.

[7] F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Proc. conference on Formal Methods in Programming and their Applications*, pages 128–141. Springer, 1993.

[8] G. Bracha. Pluggable type systems. In *Proc. Workshop on Revival of Dynamic Languages*, 2004.

[9] N. R. Cameron, S. Drossopoulou, and E. Ernst. A model for java with wildcards. In *Proc. ECOOP*, volume 5142 of *LNCS*, pages 2–26. Springer, 2008.

[10] P. Chalin and P. R. James. Non-null references by default in Java: Alleviating the nullity annotation burden. In *Proc. ECOOP*, pages 227–247. Springer, 2007.

[11] B. Chin, S. Markstrum, and T. Millstein. Semantic type qualifiers. In *Proc. PLDI*, pages 85–95. ACM Press, 2005.

[12] B. Chin, S. Markstrum, T. Millstein, and J. Palsberg. Inference of user-defined type qualifiers and qualifier rules. In *Proc. ESOP*, 2006.

[13] J.-D. Choi, D. Grove, M. Hind, and V. Sarkar. Efficient and precise modeling of exceptions for the analysis of java programs. In *Proc. PASTE*, pages 21–31, 1999.

[14] M. Cielecki, J. Fulara, K. Jakubczyk, and L. Jancewicz. Propagation of JML non-null annotations in Java programs. In *Proc. PPPJ*, pages 135–140. ACM Press, 2006.

[15] V. Cremet, F. Garillot, S. Lenglet, and M. Odersky. A core calculus for scala type checking. In *Proceedings of the Mathematical Foundations of Computer Science*, volume 4162 of *Lecture Notes in Computer Science*, pages 1–23. Springer, 2006.

[16] S. Drossopoulou and S. Eisenbach. Java is type safe - probably. In *Proc. ECOOP*, pages 389–418, 1997.

[17] T. Ekman and G. Hedin. Pluggable checking and inferencing of non-null types for Java. *Journal of Object Technology*, 6(9):455–475, 2007.

[18] M. Ernst. Type annotations specification (JSR) 308, 2009.

[19] M. Fähndrich, J. S. Foster, Z. Su, and A. Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Proc. PLDI*, pages 85–96. ACM Press, 1998.

[20] M. Fähndrich and K. R. M. Leino. Declaring and checking non-null types in an object-oriented language. In *Proc. OOPSLA*, pages 302–312. ACM Press, 2003.

[21] C. Fecht and H. Seidl. An even faster solver for general systems of equations. In *Proc. Static Analysis Symposium*, pages 189–204. Springer, 1996.

[22] C. Fecht and H. Seidl. Propagating differences: An efficient new fixpoint algorithm for distributive constraint systems. In *Proc. ESOP*, pages 90–104. Springer, 1998.

[23] M. Flatt, S. Krishnamurthi, and M. Felleisen. A programmer's reduction semantics for classes and mixins. In *Formal Syntax and Semantics of Java*, pages 241–269. Springer-Verlag, 1999.

[24] J. S. Foster, M. Fähndrich, and A. Aiken. A theory of type qualifiers. In *Proc. PLDI*, pages 192–203. ACM Press, 1999.

[25] J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *Proc. PLDI*, pages 1–12. ACM Press, 2002.

[26] E. Gagnon, L. J. Hendren, and G. Marceau. Efficient inference of static types for java bytecode. In *Proc. SAS*, volume 1824, pages 199–219. Springer, 2000.

[27] J. Gosling, G. S. B. Joy, and G. Bracha. *The Java Language Specification, 3rd Edition*. Prentice Hall, 2005.

[28] D. Greenfieldboyce and J. S. Foster. Type qualifier inference for java. In *Proc. OOPSLA*, pages 321–336. ACM Press, 2007.

[29] S. Horwitz, A. J. Demers, and T. Teitelbaum. An efficient general iterative algorithm for dataflow analysis. *Acta Informatica*, 24(6):679–694, 1987.

[30] L. Hubert, T. P. Jensen, and D. Pichardie. Semantic foundations and inference of non-null annotations. In *Proceedings of the Formal Methods for Open Object-Based Distributed Systems*, volume 5051 of *LNCS*, pages 132–149. Springer, 2008.

[31] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *TOPLAS*, 23(3):396–459, May 2001.

[32] J. Jorgensen. Improving the precision and correctness of exception analysis in SOOT. Technical report, McGill University, Canada, 2003.

[33] C. League, Z. Shao, and V. Trifonov. Type-preserving compilation of Featherweight IL. In *Proc. Workshop on Formal Techniques for Java-like Programs*, 2002.

[34] X. Leroy. Java bytecode verification: algorithms and formalizations. *Journal of Automated Reasoning*, 30(3/4):235–269, 2003.

[35] O. Lhoták and L. J. Hendren. Context-sensitive points-to analysis: Is it worth it? In *Proc. Compiler Construction*, pages 47–64. Springer, 2006.

[36] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison Wesley, second edition, 1999.

[37] C. Male, D. J. Pearce, A. Potanin, and C. Dymnikov. Java bytecode verification for @nonnull types. In *Proc. Compiler Construction*, pages 229–244, 2008.

[38] J. Miecznikowski and L. Hendren. Decompiling Java bytecode: Problems, traps and pitfalls. In *Proc. Compiler Construction*, pages 153–184, 2002.

[39] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. POPL*, pages 228–241, 1999.

[40] J. Palsberg and M. I. Schwartzbach. Object-oriented type inference. In *Proc. OOPSLA*, pages 146–161. ACM Press, 1991.

[41] D. J. Pearce, P. H. J. Kelly, and C. Hankin. Online cycle detection and difference propagation: Applications to pointer analysis. *Software Quality Journal*, 12(4):309–335, 2004.

[42] D. J. Pearce, P. H. J. Kelly, and C. Hankin. Efficient field-sensitive pointer analysis for C. *ACM Transactions on Programming Languages and Systems*, 30(1), 2008.

[43] J. Plevyak and A. A. Chien. Precise concrete type inference for object-oriented languages. In *Proc. OOPSLA*, pages 324–340. ACM Press, 1994.

[44] P. Pominville, F. Qian, R. Vallée-Rai, L. Hendren, and C. Verbrugge. A framework for optimizing Java using attributes. In *Proc. Compiler Construction*, pages 334–554, 2001.

[45] A. Potanin, J. Noble, D. Clarke, and R. Biddle. Generic ownership for generic java. In *Proc. OOPSLA*, pages 311–324. ACM Press, 2006.

[46] A. Rountev, A. Milanova, and B. G. Ryder. Points-to analysis for Java using annotated constraints. In *Proc. OOPSLA*, pages 43–55. ACM Press, 2001.

[47] M. S. Tschantz and M. D. Ernst. Javari: adding reference immutability to Java. In *Proc. OOPSLA*, pages 211–230. ACM Press, 2005.

[48] R. Vallée-Rai, E. Gagnon, L. J. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *Proc. Compiler Construction*, pages 18–34. Springer-Verlag, 2000.

[49] T. Wang and S. F. Smith. Precise constraint-based type inference for Java. In *Proc. ECOOP*, pages 99–117. Springer, 2001.

[50] T. Zhao, J. Palsber, and J. Vite. Lightweight confinement for featherweight Java. In *Proc. OOPSLA*, pages 135–148. ACM Press, Nov. 2003.