# A Calculus for Constraint-Based Flow Typing

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

## ABSTRACT

Flow typing offers an alternative to traditional Hindley-Milner type inference. A key distinction is that variables may have different types at different program points. Flow typing systems are typically formalised in the style of a dataflow analysis. In the presence of loops, this requires a fix-point computation over typing environments. Unfortunately, for some flow typing problems, the standard iterative fix-point computation may not terminate. We formalise such a problem we encountered in developing the Whiley programming language, and present a novel constraint-based solution which is guaranteed to terminate. This provides a foundation for others when developing such flow typing systems.

## Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: Object-oriented Programming; D.2.4 [**Software/Program Verification**]: Formal Methods; F.4.1 [**Mathematical Logic**]: Lambda calculus and related systems

## General Terms

Languages, Theory

## Keywords

Type Theory, Structural Typing, Flow Typing

## 1. INTRODUCTION

Type inference is useful for simplifying and reasoning about statically typed languages. Scala, C#3.0, OCaml all employ local type inference (in some form) to reduce syntactic overhead. Type inference can also be used to type existing untyped programs (e.g. in JavaScript [12] or Python [5]). Traditional type inference in the style of Hindley-Milner requires exactly one type be inferred for each program variable. Flow typing offers an alternative where a variable may have different types at different program points. The technique is adopted from flow-sensitive program analysis and has been used for non-null types [17], purity checking [23], information flow [18, 14, 27], and more [8, 28]. Few languages exist which incorporate flow typing directly. Typed Racket [28] provides a *typed* sister language for *untyped* Racket, where flow typing is essential to capture common idioms in the untyped language. Similarly, the Whiley language employs flow typing to give it the look-and-feel of a dynamically typed language [15, 25]. Finally, Groovy 2.0 has very recently incorporated an optional flow typing system [11].

A defining characteristic of flow typing is the ability to *retype* a variable — that is, assign it a completely unrelated type. The JVM Bytecode Verifier provides an excellent illustration:

```
public static void f(int):
  iload 0     // load register 0 on stack
  i2f         // convert int to float
  fstore 0    // store float to register 0
  ...
```

In the above, register 0 contains the parameter value on entry and, initially, has type **int**. The type of register 0 is subsequently changed to `float` by the `fstore` bytecode. To ensure type safety, the JVM bytecode verifier employs a typing algorithm based upon dataflow analysis [16]. This tracks the type of a variable at each program point, allowing it easily to handle the above example.

### 1.1 Contributions

Existing flow typing systems are generally formulated in the style of a dataflow analysis (e.g. [16, 17]). In the presence of loops, this requires a fix-point computation over typing environments. Unfortunately, for some flow typing problems, the standard iterative fix-point computation may not terminate. We formalise such a problem that we encountered in developing the Whiley programming language [15, 25], and present a novel constraint-based solution guaranteed to terminate. Finally, whilst our language of constraints is similar to previous constraint-based type inference systems (e.g. [22, 2, 29, 6]), the key novelty of our approach lies in a mechanism for extracting recursive types from constraints via elimination and substitution.

## 2. SYNTAX, SEMANTICS & SUBTYPING

We now introduce our calculus, called *FT* (for Flow-Typing), which is specifically kept to a minimum to focus on the interesting problem. The following gives a syntactic definition of types in FT:

$$\mathtt{T} ::= \mathtt{void} \mid \mathtt{any} \mid \mathtt{int} \mid \{\mathtt{T}_1\ \mathtt{f}_1, \ldots, \mathtt{T}_n\ \mathtt{f}_n\} \mid \mathtt{T}_1 \vee \mathtt{T}_2 \mid \mu\mathtt{X}.\mathtt{T} \mid \mathtt{X}$$

Here, **void** represents the empty set of values (i.e. $\bot$), whilst **any** the set of all possible values (i.e. $\top$). Also, $\{\mathtt{T}_1\ \mathtt{f}_1, \ldots, \mathtt{T}_n\ \mathtt{f}_n\}$ represents a record with one or more fields. The union $\mathtt{T}_1 \vee \mathtt{T}_2$ is a type whose values are in $\mathtt{T}_1$ *or* $\mathtt{T}_2$. Union types are used to characterise information flow at meet points in the control-flow graph.

Types of the form $\mu$X.T describe *recursive* data structures. For example, $\mu\mathtt{X}.(\{\mathtt{int\ data}\} \vee \{\mathtt{int\ data}, \mathtt{X\ next}\})$ gives the type of a linked list, whilst $\mu\mathtt{X}.(\{\mathtt{int\ data}\} \vee \{\mathtt{int\ data}, \mathtt{X\ lhs}, \mathtt{X\ rhs}\})$ gives the type of a binary tree. For simplicity, recursive types are treated in an *equi-recursive* fashion [26]. That is, recursive types and their unfoldings are not distinguished. For example, the recursive type $\mu\mathtt{X}.(\mathtt{int} \vee \{\mathtt{int\ data}, \mathtt{X\ next}\})$ and its one-step unfolding $\mathtt{int} \vee \{\mathtt{int\ data}, \mu\mathtt{X}.(\mathtt{int} \vee \{\mathtt{int\ data}, \mathtt{X\ next}\})\ \mathtt{next}\}$ are considered identical, and so on. Thus, we don't need to handle recursive types explicitly as, whenever we encounter $\mu$X.T, we implicitly unfold it to $\mathtt{T}[\mathtt{X} \mapsto \mu\mathtt{X}.\mathtt{T}]$ as necessary.

## 2.1 Subtyping

The subtyping rules are given in Figure 1 and employ judgements of the form "$\mathtt{T}_1 \leq \mathtt{T}_2 \lfloor \mathcal{C}$", read as: $\mathtt{T}_1$ is a subtype of $\mathtt{T}_2$ under assumptions $\mathcal{C}$.

Definition 1 (Subtyping). *Let* $\mathtt{T}_1$ *and* $\mathtt{T}_2$ *be types. Then,* $\mathtt{T}_1$ *is a subtype of* $\mathtt{T}_2$*, denoted* $\mathtt{T}_1 \leq \mathtt{T}_2$*, iff* $\mathtt{T}_1 \leq \mathtt{T}_2 \lfloor \emptyset$.

The set of assumptions $\mathcal{C}$ helps ensure the subtype rules from Figure 1 terminate. The S-I rule is critical here as it protects against infinite recursion, following the standard treatment of recursive types (see e.g. [26, 10]). Apart from assumption sets, the rules of Figure 1 are mostly straightforward. Subtyping of records is via rule S-R which allows for *depth* but (for simplicity) not *width* [26]. Thus, it follows that $\{\mathtt{T}_1\ \mathtt{f}_1, \ldots, \mathtt{T}_n\ \mathtt{f}_n\} \leq \{\mathtt{T}'_1\ \mathtt{g}_1, \ldots, \mathtt{T}'_m\ \mathtt{g}_m\}$ if $\mathtt{n} = \mathtt{m}$ and $\forall 1 \leq \mathtt{i} \leq \mathtt{n}.(\mathtt{f}_i = \mathtt{g}_i \wedge \mathtt{T}_i \leq \mathtt{T}'_i)$ (i.e. both records have the same fields and each field in the former subtypes its corresponding field in the latter). Note, it is safe for e.g. $\{\mathtt{int\ f}\} \leq \{\mathtt{any\ f}\}$ to hold because types in FT are not *reference* types (as in e.g. Java), but *value* types. Rule S-U3 is perhaps the most interesting, as it captures distributivity over records. For example, it follows under S-U3 that $\{\mathtt{int} \vee \{\mathtt{int\ x}\}\ \mathtt{f}\} \leq \{\mathtt{int\ f}\} \vee \{\{\mathtt{int\ x}\}\ \mathtt{f}\}$.

Finally, FT's subtype relation forms a *join-semi lattice*. That is, any two types $\mathtt{T}_1, \mathtt{T}_2$ have a well defined *least upper bound* (denoted $\mathtt{T}_1 \sqcup \mathtt{T}_2$). This is trivially true since it corresponds to $\mathtt{T}_1 \vee \mathtt{T}_2$.

## 2.2 Syntax

Figure 2 gives the syntax of FT where $[\![ \cdot ]\!]^\ell$ is not part of the syntax but (following [19]) identifies the distinct program points and associates each with a unique label $\ell$ (these will be explained later). An example FT program is given below:

```
int f(int x) {
  y = 1¹
  z = {f : 1}²
  while x < y³ { x = z.f⁴ }
  return x⁵
}
```

Whilst FT programs are fairly limited, they characterise an interesting flow typing problem which cannot easily be solved using an iterative fix-point computation (such as is commonly used for dataflow analysis).

## 2.3 Semantics

A small-step operational semantics for FT is given in Figure 3. The semantics describe an abstract machine executing statements of the program and (hopefully) halting to produce a value. Here, $\Delta$ is the *runtime environment*, whilst v denotes *runtime values*. A runtime environment $\Delta$ maps variables to their current runtime value.

In Figure 3, `halt(v)` is used to indicate the machine has halted producing value v. This must be distinguished from the notion of

**Subtyping:**

$$\frac{}{\mathtt{T} \leq \mathtt{T} \lfloor \mathcal{C}} \qquad \frac{\{\mathtt{T}_1 \leq \mathtt{T}_2\} \subseteq \mathcal{C}}{\mathtt{T}_1 \leq \mathtt{T}_2 \lfloor \mathcal{C}} \qquad \text{(S-F, S-I)}$$

$$\frac{}{\mathtt{void} \leq \mathtt{T} \lfloor \mathcal{C}} \qquad \frac{}{\mathtt{T} \leq \mathtt{any} \lfloor \mathcal{C}} \qquad \text{(S-V, S-A)}$$

$$\frac{\begin{array}{c}\mathcal{C}_2 = \mathcal{C}_1 \cup \{\mathtt{T} \leq \mathtt{T}'\} \\ \mathtt{T}_1 \leq \mathtt{T}'_1 \lfloor \mathcal{C}_2 \ \ldots\ \mathtt{T}_n \leq \mathtt{T}'_n \lfloor \mathcal{C}_2 \\ \mathtt{T} = \{\mathtt{T}_1\ \mathtt{f}_1, \ldots, \mathtt{T}_n\ \mathtt{f}_n\} \quad \mathtt{T}' = \{\mathtt{T}'_1\ \mathtt{f}_1, \ldots, \mathtt{T}'_n\ \mathtt{f}_n\}\end{array}}{\mathtt{T} \leq \mathtt{T}' \lfloor \mathcal{C}_1} \qquad \text{(S-R)}$$

$$\frac{\begin{array}{c}\mathcal{C}_2 = \mathcal{C}_1 \cup \{\mathtt{T}_1 \leq \mathtt{T}_2 \vee \mathtt{T}_3\} \\ \exists \mathtt{i} \in \{2,3\}.\mathtt{T}_1 \leq \mathtt{T}_i \lfloor \mathcal{C}_2\end{array}}{\mathtt{T}_1 \leq \mathtt{T}_2 \vee \mathtt{T}_3 \lfloor \mathcal{C}_1} \qquad \text{(S-U1)}$$

$$\frac{\begin{array}{c}\mathcal{C}_2 = \mathcal{C}_1 \cup \{\mathtt{T}_1 \vee \mathtt{T}_2 \leq \mathtt{T}_3\} \\ \mathtt{T}_1 \leq \mathtt{T}_3 \lfloor \mathcal{C}_2 \quad \mathtt{T}_2 \leq \mathtt{T}_3 \lfloor \mathcal{C}_2\end{array}}{\mathtt{T}_1 \vee \mathtt{T}_2 \leq \mathtt{T}_3 \lfloor \mathcal{C}_1} \qquad \text{(S-U2)}$$

$$\frac{\begin{array}{c}\mathtt{T} = \{\mathtt{T}_1\ \mathtt{f}_1 \ldots, \mathtt{T}_i \vee \mathtt{T}'_i\ \mathtt{f}_i \ldots, \mathtt{T}_n\ \mathtt{f}_n\} \\ \mathtt{S}_1 = \{\mathtt{T}_1\ \mathtt{f}_1 \ldots, \mathtt{T}_i\ \mathtt{f}_i, \ldots, \mathtt{T}_n\ \mathtt{f}_n\} \\ \mathtt{S}_2 = \{\mathtt{T}_1\ \mathtt{f}_1 \ldots, \mathtt{T}'_i\ \mathtt{f}_i, \ldots, \mathtt{T}_n\ \mathtt{f}_n\}\end{array}}{\mathtt{T} \leq \mathtt{S}_1 \vee \mathtt{S}_2 \lfloor \mathcal{C}} \qquad \text{(S-U3)}$$

**Figure 1: Subtyping rules for FT.**

**Syntax:**

$$\begin{array}{lll}
\mathtt{F} & ::= & \mathtt{T\ f}(\mathtt{T}_1\ \mathtt{n}_1, \ldots, \mathtt{T}_n\ \mathtt{n}_n)\ \{\mathtt{B}\} \\
\mathtt{B} & ::= & \mathtt{S}\ \mathtt{B} \mid \epsilon \\
\mathtt{S} & ::= & [\![\mathtt{n} = \mathtt{v}]\!]^\ell \mid [\![\mathtt{n} = \mathtt{m}]\!]^\ell \mid [\![\mathtt{n.f} = \mathtt{m}]\!]^\ell \mid [\![\mathtt{n} = \mathtt{m.f}]\!]^\ell \\
 & & \mid [\![\mathtt{return\ n}]\!]^\ell \mid \mathtt{while}\ [\![\mathtt{n} < \mathtt{m}]\!]^\ell\ \{\mathtt{B}\} \\
\mathtt{v} & ::= & \{\mathtt{f}_1 : \mathtt{v}_1, \ldots, \mathtt{f}_n : \mathtt{v}_n\} \mid \mathtt{i}
\end{array}$$

**Figure 2: Syntax for FT. Here, n, m represent variable identifiers, whilst i represents the integer constants.**

**Semantics:**

$$\frac{}{\langle \Delta,\ [\![\mathtt{n}{=}\mathtt{v}]\!]^\ell\ \mathtt{B}\rangle \longrightarrow \langle \Delta[\mathtt{n}{\mapsto}\mathtt{v}],\ \mathtt{B}\rangle} \qquad \text{(R-VC)}$$

$$\frac{\mathtt{v} = \Delta(\mathtt{m})}{\langle \Delta,\ [\![\mathtt{n}{=}\mathtt{m}]\!]^\ell\ \mathtt{B}\rangle \longrightarrow \langle \Delta[\mathtt{n}{\mapsto}\mathtt{v}],\ \mathtt{B}\rangle} \qquad \text{(R-VV)}$$

$$\frac{\Delta(\mathtt{m}) = \{\ldots, \mathtt{f} : \mathtt{v}, \ldots\}}{\langle \Delta,\ [\![\mathtt{n}{=}\mathtt{m.f}]\!]^\ell\ \mathtt{B}\rangle \longrightarrow \langle \Delta[\mathtt{n}{\mapsto}\mathtt{v}],\ \mathtt{B}\rangle} \qquad \text{(R-VF)}$$

$$\frac{\begin{array}{c}\Delta(\mathtt{n}) = \{\mathtt{f}_1 : \mathtt{v}_1, \ldots, \mathtt{f}_n : \mathtt{v}_n\} \\ \mathtt{v} = \Delta(\mathtt{n})[\mathtt{f} \mapsto \Delta(\mathtt{m})]\end{array}}{\langle \Delta,\ [\![\mathtt{n.f}{=}\mathtt{m}]\!]^\ell\ \mathtt{B}\rangle \longrightarrow \langle \Delta[\mathtt{n}{\mapsto}\mathtt{v}],\ \mathtt{B}\rangle} \qquad \text{(R-FV)}$$

$$\frac{\mathtt{v} = \Delta(\mathtt{n})}{\langle \Delta,\ [\![\mathtt{return\ n}]\!]^\ell\ \mathtt{B}\rangle \longrightarrow \mathtt{halt}(\mathtt{v})} \qquad \text{(R-RV)}$$

$$\frac{\Delta(\mathtt{n}) < \Delta(\mathtt{m})}{\begin{array}{c}\langle \Delta, \mathtt{while}\ [\![\mathtt{n}{<}\mathtt{m}]\!]^\ell\ \{\mathtt{B}_1\}\ \mathtt{B}_2\rangle \\ \longrightarrow \langle \Delta,\ \mathtt{B}_1\ \mathtt{while}\ [\![\mathtt{n}{<}\mathtt{m}]\!]^\ell\ \{\mathtt{B}_1\}\ \mathtt{B}_2\rangle\end{array}} \qquad \text{(R-W1)}$$

$$\frac{\Delta(\mathtt{n}) \geq \Delta(\mathtt{m})}{\langle \Delta, \mathtt{while}\ [\![\mathtt{n}{<}\mathtt{m}]\!]^\ell\ \{\mathtt{B}_1\}\ \mathtt{B}_2\rangle \longrightarrow \langle \Delta,\ \mathtt{B}_2\rangle} \qquad \text{(R-W2)}$$

**Figure 3: Small-step operational semantics for statements in FT.**

being "stuck". The latter occurs when the machine has not halted, but cannot execute further (because none of the transition rules from Figure 3 applies). For example, a statement $\mathtt{n} = \mathtt{m.f}$ can result in the machine being stuck. To see why, notice that only rule R-VF can be applied to such a statement. This has an explicit requirement that `m` currently holds a record value containing at least field `f`. Thus, in the case that `m` does not currently hold a record value, or that it holds a record value which does not contain a field `f`, then the machine will be stuck.

Some observations can be made from Figure 3. Firstly, *variables do not need to be explicitly declared* — rather, they are declared implicitly by assignment. Secondly, *variables must be defined before being used* — as, otherwise, the machine will get stuck. Finally, assignments to fields succeed *even when the assigned field doesn't exist*. This is captured in rule R-FV, where the record value being assigned is updated with a (potentially new) field `f`. For example:

```
{any f, int g} f(any y) {
     x = {f : 1}^1
     x.f = y^2
     x.g = 1^3
     return x^4
}
```

This program executes under the rules of Figure 3 without getting stuck. Furthermore, as we will see, it can be type checked with appropriate flow typing rules (§4). The key to this is that variable `x` has different types at different program points: after initialisation, it has type $\{\mathtt{int\ f}\}$; after the subsequent assignment to field `f` this becomes $\{\mathtt{any\ f}\}$; and, finally, after the assignment to field `g` it has type $\{\mathtt{any\ f}, \mathtt{int\ g}\}$.

The ability to safely update field types in FT contrasts with traditional object-oriented languages (e.g. Java) where assignments must respect the declared type of the assigned field. The semantics of FT are (in some ways) closer to those of a dynamically typed language where one can assign to fields and variables at will.

## 3. DATAFLOW-BASED FLOW TYPING

We now formulate the typing rules for FT as a *dataflow analysis*. This is an intuitive and commonly used approach (e.g. [16, 17]). Our purpose is to highlight an inherent limitation of using this approach for FT — namely, that it requires finding a fix-point over typing environments for which the standard iterative fix-point computation fails to terminate in some cases.

Dataflow-based flow typing requires a separate environment, $\Gamma^\ell$, for each program point $\ell$. This gives the types of all variables immediately before the statement at $\ell$. For example, consider a small program (left) along with its typing environments (right):

```
int f(int x) {
     y = x^1       // Γ^1 = {x ↦ int}
     return y^2  // Γ^2 = {x ↦ int, y ↦ int}
}
```

Since $y$ is defined on line 1, it is absent from $\Gamma^1$ (which represents the environment immediately *before* line 1). Now, consider:

```
int ∨ {int g} f(int x) {
     y = 1^1
     while x < x^2 { y = {g : 1}^3 }
     return y^4
}
```

**Function Typing (dataflow):**

$$\frac{\{\mathtt{n_1} \mapsto \mathtt{T_1}, \ldots, \mathtt{n_k} \mapsto \mathtt{T_k}, \$ \mapsto \mathtt{T}\} \vdash \mathtt{B} : \Gamma}{\vdash \mathtt{T\ f}(\mathtt{T_1\ n_1}, \ldots, \mathtt{T_k\ n_k})\ \{\mathtt{B}\}} \qquad \text{(T-FUN)}$$

**Block Typing (dataflow):**

$$\frac{\Gamma_0 \vdash \mathtt{S} : \Gamma_1 \quad \Gamma_1 \vdash \mathtt{B} : \Gamma_2}{\Gamma_0 \vdash \mathtt{S\ B} : \Gamma_2} \qquad \text{(T-BLK)}$$

**Statement Typing (dataflow):**

$$\frac{\vdash \mathtt{v} : \mathtt{T}}{\Gamma \vdash [\![\mathtt{n}=\mathtt{v}]\!]^\ell : \Gamma[\mathtt{n} \mapsto \mathtt{T}]} \quad \frac{\Gamma(\mathtt{m}) = \mathtt{T}}{\Gamma \vdash [\![\mathtt{n}=\mathtt{m}]\!]^\ell : \Gamma[\mathtt{n} \mapsto \mathtt{T}]} \qquad \text{(T-VC, T-VV)}$$

$$\frac{\Gamma(\mathtt{m}) = \{\ldots, \mathtt{T\ f}, \ldots\}}{\Gamma \vdash [\![\mathtt{n}=\mathtt{m.f}]\!]^\ell : \Gamma[\mathtt{n} \mapsto \mathtt{T}]} \quad \frac{\begin{array}{c}\Gamma(\mathtt{n}) = \{\mathtt{T_1\ f_1}, \ldots, \mathtt{T_n\ f_n}\} \\ \mathtt{T} = \Gamma(\mathtt{n})[\mathtt{f} \mapsto \Gamma(\mathtt{m})]\end{array}}{\Gamma \vdash [\![\mathtt{n.f}=\mathtt{m}]\!]^\ell : \Gamma[\mathtt{n} \mapsto \mathtt{T}]} \qquad \text{(T-VF, T-FV)}$$

$$\frac{\Gamma(\mathtt{n}) \leq \Gamma(\$)}{\Gamma \vdash [\![\mathtt{return\ n}]\!]^\ell : \emptyset} \qquad \text{(T-RV)}$$

$$\frac{\begin{array}{c}\Gamma_0 \sqcup \Gamma_1 \vdash \mathtt{B} : \Gamma_1 \\ \Gamma_0 \sqcup \Gamma_1(\mathtt{n}) = \mathtt{int} \quad \Gamma_0 \sqcup \Gamma_1(\mathtt{m}) = \mathtt{int}\end{array}}{\Gamma_0 \vdash \mathtt{while}\ [\![\mathtt{n} < \mathtt{m}]\!]^\ell\ \{\mathtt{B}\} : \Gamma_0 \sqcup \Gamma_1} \qquad \text{(T-WHILE)}$$

**Figure 4: Dataflow-based typing rules for FT.**

The question is, what type does `y` have in $\Gamma^4$? We know that `y` has type `int` if the loop isn't taken, or $\{\mathtt{int\ g}\}$ otherwise. To capture this, we compute the *least upper bound* of the type environments:

$$\begin{array}{l}\Gamma^4 = \{\mathtt{x} \mapsto \mathtt{int}, \mathtt{y} \mapsto \mathtt{int}\} \sqcup \{\mathtt{x} \mapsto \mathtt{int}, \mathtt{y} \mapsto \{\mathtt{int\ g}\}\} \\ \quad \hookrightarrow \{\mathtt{x} \mapsto \mathtt{int}, \mathtt{y} \mapsto \mathtt{int} \vee \{\mathtt{int\ g}\}\}\end{array}$$

Here, $\Gamma^4(\mathtt{y}) = \mathtt{int} \vee \{\mathtt{int\ g}\}$ as an `int` value can flow from *before* the loop, whilst $\{\mathtt{int\ g}\}$ can flow from *around* the loop. Here, we are tacitly assuming the loop can be executed zero or more times, even though (in principle) we could be more precise. This follows the standard approach used in dataflow analysis (see e.g. [19]).

Definition 2 (Environment Subtyping). *Let $\Gamma^{\ell 1}$ and $\Gamma^{\ell 2}$ be typing environments. Then, we say that $\Gamma^{\ell 1}$ subtypes $\Gamma^{\ell 2}$, denoted $\Gamma^{\ell 1} \leq \Gamma^{\ell 2}$, iff $\forall \mathtt{x} \in \textbf{dom}(\Gamma^{\ell 2}).\Gamma^{\ell 1}(\mathtt{x}) \leq \Gamma^{\ell 2}(\mathtt{x})$.*

For example, the following hold under Definition 2:

$$\begin{array}{rcl}\{\mathtt{x} \mapsto \mathtt{int}\} & \leq & \{\mathtt{x} \mapsto \mathtt{any}\} \\ \{\mathtt{x} \mapsto \{\mathtt{int\ f}\}, \mathtt{y} \mapsto \mathtt{int}\} & \leq & \{\mathtt{x} \mapsto \mathtt{any}\}\end{array}$$

Since the underlying subtype relation over types forms a join semilattice, it follows that environment subtyping does as well (where $\perp = \emptyset$ and $\top$ maps all program variables to `any`). Hence, it follows that any two environments have a unique least upper bound.

### 3.1 Dataflow-Based Typing Rules

The typing rules for statements describe their *effect* on the typing environment. They are judgements of the form $\Gamma \vdash \mathtt{S} : \Gamma'$ where $\Gamma$ represents the environment immediately before `S`, and $\Gamma'$ represents that immediately after. For example, consider:

```
int f(any x) { x = 1^1 ; return x^2 }
```

Here, $\Gamma^1 = \{\mathtt{x} \mapsto \mathtt{any}, \$ \mapsto \mathtt{int}\}$ gives the environment immediately before the assignment. Then, the typing environment obtained immediately after it is $\Gamma^2 = \{\mathtt{x} \mapsto \mathtt{int}, \$ \mapsto \mathtt{int}\}$.

The dataflow-based typing rules for FT are given in Figure 4. Rule T-FUN states that an FT function can be typed if its body can be typed with parameters mapped to their declared types. The special variable \$ is included to provide access to the return type. Rule T-BLK threads an environment through a sequence of statements.

Rule T-VC exploits the fact that values have fixed types (obtained via $\vdash \mathtt{v} : \mathtt{T}$). The requirement $\Gamma(\mathtt{m}) = \{\ldots, \mathtt{T}\ \mathtt{f}, \ldots\}$ in rule T-VF ensures that m holds a record containing field f at the given point. Similarly, in T-VF, $\{\mathtt{T}_1\ \mathtt{f}_1, \ldots, \mathtt{T}_n\ \mathtt{f}_n\}[\mathtt{f} \mapsto \mathtt{T}]$ constructs a type identical to $\{\mathtt{T}_1\ \mathtt{f}_1, \ldots, \mathtt{T}_n\ \mathtt{f}_n\}$, but where field f now has type T (even if the original didn't contain a field f). Rule T-RV confirms the returned value is a subtype of the declared return type. Finally, rule T-WHILE requires a fix-point be found for the environment produced from the body, and we discus this in more detail below.

## 3.2 Termination

Computing a fix-point for a dataflow analysis is normally done using an iterative computation (see e.g. [19]). Unfortunately, using such a computation to solve the typing rules of Figure 4 will not always terminate. The following illustrates:

```
void loopy(int x, int y) {
  z = {f:1}^1 ; while x < y^2 { z.f = z^3 }
}
```

This example causes an iterative fix-point solver for rule T-WHILE to iterate forever, generating larger and larger environments:

$$\begin{aligned}
\Gamma^3 &= \{\mathtt{z} \mapsto \{\mathtt{int}\ \mathtt{f}\}, \ldots\}\\
\Gamma^3 &= \{\mathtt{z} \mapsto \{\mathtt{int} \vee \{\mathtt{int}\ \mathtt{f}\}\ \mathtt{f}\}, \ldots\}\\
\Gamma^3 &= \{\mathtt{z} \mapsto \{\mathtt{int} \vee \{\mathtt{int}\ \mathtt{f}\} \vee \{\mathtt{int} \vee \{\mathtt{int}\ \mathtt{f}\}\ \mathtt{f}\}\ \mathtt{f}\}, \ldots\}\\
\ldots
\end{aligned}$$

Proving that an iterative fix-point computation always terminates is normally done by showing two key properties: firstly, the domain (i.e. types) and partial order (i.e. subtyping) must form a *join semi-lattice* (of finite height); secondly, the transfer functions (i.e. the rules of Figure 4) must be *monotonic*. Sadly, the lattice of types in FT has infinite height, hence such a proof can't apply. Observe, however, *that intuitively a valid typing of the above example exists*:

$$\Gamma^3 = \{\mathtt{x} \mapsto \mathtt{int}, \mathtt{y} \mapsto \mathtt{int}, \mathtt{z} \mapsto \mu\mathtt{X}.\{(\mathtt{int} \vee \mathtt{X})\ \mathtt{f}\}\} \qquad (1)$$

The key problem, then, is how one could obtain such a typing in practice. In fact, there are many examples in the dataflow analysis literature of systems with lattices of infinite height (e.g. *integer range analysis* [19]). Such systems are forced to terminate through the introduction of a *widening operator*. Such an operator is applied after a certain number of iterations of the computation. Typically, it will attempt to "guess" a value which causes the computation to converge and, if that fails, will move to a worst-case default (e.g. $\Gamma^3 = \{\mathtt{x} \mapsto \mathtt{int}, \mathtt{y} \mapsto \mathtt{int}, \mathtt{z} \mapsto \mathtt{any}\}$ — which in this case prevents the program from being typed).

## 4. CONSTRAINT-BASED FLOW TYPING

We now present a novel *constraint-based* formulation of the typing rules for FT in the style of e.g. [13, 2]. Critically, this does not require a fix-point computation and, hence, is guaranteed to terminate. Our language of type constraints is as follows:

$$\begin{aligned}
\mathtt{c} &::= \mathtt{n}_\ell \sqsupseteq \mathtt{e} \mid \mathtt{T} \sqsupseteq \mathtt{e}\\
\mathtt{e} &::= \mathtt{T} \mid \mathtt{n}_\ell \mid \mathtt{e.f} \mid \mathtt{e}_1[\mathtt{f} \mapsto \mathtt{e}_2] \mid \bigsqcup \mathtt{e}_i
\end{aligned}$$

Here, T represents a fixed type from those outlined in §2, whilst $\mathtt{n}_\ell$ denotes the set of *labelled* type variables which range over types (though, for simplicity, we will sometimes omit the label). Furthermore, e represents *constraint expressions* which are used to build up the right-hand side of a constraint. Finally, $\mathtt{e}_1[\mathtt{f} \mapsto \mathtt{e}_2]$ can be viewed as updating the record returned by $\mathtt{e}_1$ such that field f now contains the value obtained from $\mathtt{e}_2$.

The idea is that, for a given FT program, we generate a set of such constraints and subsequently solve them. The following illustrates the idea:

```
int ∨ {int g} f(int x,int y){ // x0 ⊒ int,y0 ⊒ int
    r = 0^1                   // r1 ⊒ int
    while x < y^2 {           // r2 ⊒ r1 ⊔ r3
       r = {g : 1}^3          // r3 ⊒ {int g}
    }
    return r^4                // int ∨ {int g} ⊒ r2
}
```

Here, we see that each program variable may be split across multiple constraint variables (e.g. r is represented by $\mathtt{r}_1, \mathtt{r}_2$ and $\mathtt{r}_3$).

DEFINITION 3 (TYPING). *A typing, $\Sigma$, maps variables to types and* satisfies *a constraint set $\mathcal{C}$, denoted by $\Sigma \models \mathcal{C}$, if for all $\mathtt{e}_1 \sqsupseteq \mathtt{e}_2 \in \mathcal{C}$ we have $\mathcal{E}(\Sigma, \mathtt{e}_1) \geq \mathcal{E}(\Sigma, \mathtt{e}_2)$. Here, $\mathcal{E}(\Sigma, \mathtt{e})$ is the evaluation function, defined as follows:*

$$\begin{aligned}
\mathcal{E}(\Sigma, \mathtt{T}) &= \mathtt{T} && (1)\\
\mathcal{E}(\Sigma, \mathtt{n}_\ell) &= \mathtt{T} \ \ \textbf{if}\ \ \{\mathtt{n}_\ell \mapsto \mathtt{T}\} \subseteq \Sigma && (2)\\
\mathcal{E}(\Sigma, \mathtt{e.f}) &= \bigvee \mathtt{T}_i \ \ \textbf{if}\ \ \mathcal{E}(\Sigma, \mathtt{e}) = \bigvee\{\ldots, \mathtt{T}_i\ \mathtt{f}, \ldots\} && (3)\\
\mathcal{E}(\Sigma, \mathtt{e}_1[\mathtt{f} \mapsto \mathtt{e}_2]) &=\\
\bigvee\{\overline{\mathtt{T}\ \mathtt{f}}\}[\mathtt{f} \mapsto \mathtt{T}] &\ \ \textbf{if}\ \ \mathcal{E}(\Sigma, \mathtt{e}_1) = \bigvee\{\overline{\mathtt{T}\ \mathtt{f}}\}\ \textbf{and}\ \mathcal{E}(\Sigma, \mathtt{e}_2) = \mathtt{T} && (4)\\
\mathcal{E}(\Sigma, \bigsqcup \mathtt{e}_i) &= \bigvee \mathtt{T}_i \ \ \textbf{if}\ \ \mathcal{E}(\Sigma, \mathtt{e}_1) = \mathtt{T}_1, \ldots, \mathcal{E}(\Sigma, \mathtt{e}_n) = \mathtt{T}_n && (5)
\end{aligned}$$

Rule (3) selects field f from a union of one or more records containing that field (e.g. $\mathcal{E}(\emptyset, (\{\mathtt{int}\ \mathtt{f}\} \vee \{\mathtt{any}\ \mathtt{f}\}).\mathtt{f}) = \mathtt{int} \vee \mathtt{any}$). Rule (4) updates the type of field f across a union of one or more records. Here, $\bigvee\{\overline{\mathtt{T}\ \mathtt{f}}\}$ is short-hand notation for a union of records $\{\mathtt{T}^1_1\ \mathtt{f}^1_1, \ldots, \mathtt{T}^1_n\ \mathtt{f}^1_n\} \vee \ldots \vee \{\mathtt{T}^k_1\ \mathtt{f}^k_1, \ldots, \mathtt{T}^k_m\ \mathtt{f}^k_m\}$, while $\{\overline{\mathtt{T}\ \mathtt{f}}\}[\mathtt{f} \mapsto \mathtt{T}]$ constructs a type identical to $\{\overline{\mathtt{T}\ \mathtt{f}}\}$, but where field f now has type T (even if the original didn't contain a field f). Thus, it follows that $\mathcal{E}(\emptyset, (\{\mathtt{int}\ \mathtt{f}\} \vee \{\mathtt{int}\ \mathtt{g}\})[\mathtt{f} \mapsto \mathtt{any}]) = \{\mathtt{any}\ \mathtt{f}\} \vee \{\mathtt{any}\ \mathtt{f}, \mathtt{int}\ \mathtt{g}\}$.

Finally, a given FT program is considered *type safe* if a valid typing exists which satisfies the generated typing constraints.

## 4.1 Constraint-Based Typing Rules

Figure 5 gives the constraint-based typing rules for FT which have a general form of $\Gamma_0 \vdash \mathtt{S} : \Gamma_1 \downharpoonleft \mathcal{C}$ (except T-FUN, which is similar). In the constraint-based formulation, a typing environment $\Gamma$ maps each variable to the program point where its current value was defined. Finally, $\mathcal{C}$ is the constraint set which must hold (i.e. admit a valid solution) for that statement to be type safe.

As before, T-FUN initialises the typing environment from the parameter types, and adds a constraint for the return type. The latter employs a special variable, \$, to connect the return type with any returned values (via T-RV). The following illustrates:

```
int f(any x) {     // x0 ⊒ any, int ⊒ $  (T-FUN)
    x = 1^1        // x1 ⊒ int  (T-VC)
    return x^2     // $ ⊒ x1  (T-RV)
}
```

Here, $\mathtt{x}_1$ is connected to the return type through \$. Rule T-VC constrains the type of the assigned variable to that of the assigned (constant) value. The environment produced (i.e. $\Gamma[\mathtt{n} \mapsto \ell]$) equals the old (i.e. $\Gamma$) but with n mapped to $\ell$. Rule T-VV constrains the type of the assigned variable to that of the right-hand side. Here,

**Function Typing (constraints):**

$$\frac{\{\mathtt{n}^1 \mapsto 0, \ldots, \mathtt{n}^k \mapsto 0\} \vdash \mathtt{B} : \Gamma_1 \downharpoonleft \mathcal{C}_1 \quad \mathcal{C}_2 = \mathcal{C}_1 \cup \{\mathtt{n}^1_0 \sqsupseteq \mathtt{T}^1, \ldots, \mathtt{n}^k_0 \sqsupseteq \mathtt{T}^k, \mathtt{T} \sqsupseteq \$\}}{\vdash \mathtt{T}\ \mathtt{f}(\mathtt{T}^1\ \mathtt{n}^1, \ldots, \mathtt{T}^k\ \mathtt{n}^k)\ \mathtt{B} \downharpoonleft \mathcal{C}_2} \quad \text{(T-FUN)}$$

**Block Typing (constraints):**

$$\frac{\Gamma_0 \vdash \mathtt{S} : \Gamma_1 \downharpoonleft \mathcal{C}_1 \quad \Gamma_1 \vdash \mathtt{B} : \Gamma_2 \downharpoonleft \mathcal{C}_2}{\Gamma_0 \vdash \mathtt{S}\ \mathtt{B} :\ \Gamma_2 \downharpoonleft \mathcal{C}_1 \cup \mathcal{C}_2} \quad \text{(T-BLK)}$$

**Statement Typing (constraints):**

$$\frac{\vdash \mathtt{v}\ :\ \mathtt{T}}{\Gamma \vdash \llbracket \mathtt{n}{=}\mathtt{v} \rrbracket^{\ell}\ :\ \Gamma[\mathtt{n} \mapsto \ell]\ \downharpoonleft\ \{\mathtt{n}_\ell \sqsupseteq \mathtt{T}\}} \quad \text{(T-VC)}$$

$$\frac{\Gamma(\mathtt{m}) = \kappa}{\Gamma \vdash \llbracket \mathtt{n}{=}\mathtt{m} \rrbracket^{\ell}\ :\ \Gamma[\mathtt{n} \mapsto \ell]\ \downharpoonleft\ \{\mathtt{n}_\ell \sqsupseteq \mathtt{m}_\kappa\}} \quad \text{(T-VV)}$$

$$\frac{\Gamma(\mathtt{m}) = \kappa}{\Gamma \vdash \llbracket \mathtt{n}{=}\mathtt{m.f} \rrbracket^{\ell}\ :\ \Gamma[\mathtt{n} \mapsto \ell]\ \downharpoonleft\ \{\mathtt{n}_\ell \sqsupseteq \mathtt{m}_\kappa.\mathtt{f}\}} \quad \text{(T-VF)}$$

$$\frac{\Gamma(\mathtt{n}) = \kappa \quad \Gamma(\mathtt{m}) = \lambda}{\Gamma \vdash \llbracket \mathtt{n.f}{=}\mathtt{m} \rrbracket^{\ell}\ :\ \Gamma[\mathtt{n} \mapsto \ell]\ \downharpoonleft\ \{\mathtt{n}_\ell \sqsupseteq \mathtt{n}_\kappa[\mathtt{f} \mapsto \mathtt{m}_\lambda]\}} \quad \text{(T-FV)}$$

$$\frac{\Gamma(\mathtt{n}) = \kappa}{\Gamma \vdash \llbracket \mathtt{return\ n} \rrbracket^{\ell}\ :\ \emptyset\ \downharpoonleft\ \{\$ \sqsupseteq \mathtt{n}_\kappa\}} \quad \text{(T-RV)}$$

$$\frac{\begin{array}{c} \text{defs}(\mathtt{B}) = \overline{\mathtt{n}} \\ \Gamma^1 = \Gamma^0[\overline{\mathtt{n} \mapsto \ell}] \quad \Gamma^1 \vdash \mathtt{B} : \Gamma^2 \downharpoonleft \mathcal{C}_1 \\ \overline{\Gamma^0(\mathtt{n}) = \kappa} \quad \overline{\Gamma^2(\mathtt{n}) = \lambda} \\ \Gamma^1(\mathtt{n}) = \kappa \quad \Gamma^1(\mathtt{m}) = \lambda \\ \mathcal{C}_2 = \{\mathtt{int} \sqsupseteq \mathtt{n}_\kappa, \mathtt{int} \sqsupseteq \mathtt{m}_\lambda\} \\ \mathcal{C}_3 = \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\overline{\mathtt{n}_\ell \sqsupseteq \mathtt{n}_\kappa \sqcup \mathtt{n}_\lambda}\} \end{array}}{\Gamma^0 \vdash \mathtt{while}\ \llbracket \mathtt{n} < \mathtt{m} \rrbracket^{\ell}\ \{\mathtt{B}\} : \Gamma^1 \downharpoonleft \mathcal{C}_3} \quad \text{(T-WHILE)}$$

**Variable Definitions:**

$$\begin{aligned} \text{defs}(\mathtt{S}\ ;\ \mathtt{B}) &= \text{defs}(\mathtt{S}) \cup \text{defs}(\mathtt{B}) \\ \text{defs}(\llbracket \mathtt{n} = \ldots \rrbracket^{\ell}) &= \{\mathtt{n}\} \\ \text{defs}(\llbracket \mathtt{n.f} = \ldots \rrbracket^{\ell}) &= \{\mathtt{n}\} \\ \text{defs}(\llbracket \mathtt{return\ n} \rrbracket^{\ell}) &= \emptyset \\ \text{defs}(\mathtt{while}\ \llbracket \mathtt{n} < \mathtt{m} \rrbracket^{\ell}\ \{\mathtt{B}\}) &= \text{defs}(\mathtt{B}) \end{aligned}$$

**Figure 5: Constraint-Based Typing rules for FT.**

$\Gamma(\mathtt{m}) = \kappa$ determines the program point ($\kappa$) where the type variable currently representing m was defined ($\mathtt{m}_\kappa$). Rule T-VF constrains the assigned variable to the corresponding field of the right-hand side. Rule T-FV uses a constraint of the form $\mathtt{n}_\ell \sqsupseteq \mathtt{n}_\kappa[\mathtt{f} \mapsto \mathtt{m}_\lambda]$. This constrains all fields of $\mathtt{n}_\ell$ (except for f) to their corresponding type in $\mathtt{n}_\kappa$, whilst field f now maps to $\mathtt{m}_\lambda$.

Finally, in rule T-WHILE the overbar (e.g. $\overline{\mathtt{n}}$) is a short-hand indicating a list (or set) of items. The rule employs a support function, defs(B), to identify variables assigned in B. Each variable $\mathtt{n} \in \text{defs}(\mathtt{B})$ requires a constraint to merge flow from *before* the loop (i.e. $\mathtt{n}_\kappa$) with that from *around* the loop (i.e. $\mathtt{n}_\lambda$). For each, a variable $\mathtt{n}_\ell$ is created to capture this flow. This corresponds (roughly) to the placement of $\phi-$nodes in SSA form [7].

## 4.2 Variable Elimination

We now begin presenting our algorithm for solving the typing constraints generated for a given function. We first consider the *variable elimination* step. The essence is, for each variable $\mathtt{n}_\ell$, to generate a *single constraint* from which we can extract its typing.

DEFINITION 4 (VARIABLE SCOPING). *Let $\mathcal{C}_\mathcal{X}$ denote a constraint set where $\mathcal{X}$ defines the variables permissible in any constraint $\mathtt{e}_1 \sqsupseteq \mathtt{e}_2 \in \mathcal{C}_\mathcal{X}$.*

DEFINITION 5 (SINGLE ASSIGNMENT). *A constraint set $\mathcal{C}_\mathcal{X}$ is in* single assignment form *if, for each $\mathtt{n}_\ell \in \mathcal{X}$, there is at most one constraint in $\mathcal{C}_\mathcal{X}$ of the form $\mathtt{n}_\ell \sqsupseteq \mathtt{e}$.*

Any constraint set $\mathcal{C}_\mathcal{X}$ generated from the rules of Figure 5 is *almost* in single assignment form. This is because only T-RV can give rise to multiple constraints with the same left-hand side (i.e. \$). Thus, we can transform $\mathcal{C}_\mathcal{X}$ into single assignment form by collecting all such constraints and combining them:

$$\$ \sqsupseteq \mathtt{n}_{\ell 0}, \ldots, \$ \sqsupseteq \mathtt{n}_{\ell n} \implies \$ \sqsupseteq \mathtt{n}_{\ell 0} \sqcup \ldots \sqcup \mathtt{n}_{\ell n}$$

We now apply successive substitutions to eliminate variables and narrow down the final constraint for a given variable:

DEFINITION 6 (ELIMINATION STEP). *Let $\mathcal{C}_\mathcal{X}$ be a constraint set in single assignment form, where we have $\mathtt{n}_\ell \sqsupseteq \mathtt{e} \in \mathcal{C}_\mathcal{X}$. Then, we can eliminate $\mathtt{n}_\ell$ from $\mathcal{C}_\mathcal{X}$ to form a (smaller) constraint set as follows: $\mathcal{C}_{\mathcal{X}-\{\mathtt{n}_\ell\}} = \{\mathtt{e}_1 \sqsupseteq \mathtt{e}_2\llbracket \mathtt{n}_\ell \mapsto \mathtt{e} \rrbracket \mid \mathtt{e}_1 \sqsupseteq \mathtt{e}_2 \in \mathcal{C}_\mathcal{X} \wedge \mathtt{e}_1 \neq \mathtt{n}_\ell\}$.*

Here, the choice of $\mathtt{n}_\ell$ to eliminate is arbitrary. Recall that $\mathtt{e}_1$ is either a variable $\mathtt{n}_\kappa$, or a type T (i.e. not an arbitrary expression). Furthermore, $\mathtt{e}_2\llbracket \mathtt{n}_\ell \mapsto \mathtt{e} \rrbracket$ substitutes all occurrences of $\mathtt{n}_\ell$ with e in $\mathtt{e}_2$. To determine the typing for a given variable $\mathtt{n}_\ell$, we progressively eliminate variables until only $\mathtt{n}_\ell$ remains. Then, we have $\mathtt{n}_\ell \sqsupseteq \mathtt{e} \in \mathcal{C}_{\{\mathtt{n}_\ell\}}$ and from this we extract the type for $\mathtt{n}_\ell$ (discussed further in §4.3). To illustrate, we revisit our running example:

```
void loopy(int x, int y) { // $x_0 \sqsupseteq \mathtt{int}, y_0 \sqsupseteq \mathtt{int}$,
                           // $\mathtt{void} \sqsupseteq \$$ (T-FUN)
  z = {f : 1}^1            // $z_0 \sqsupseteq \{\mathtt{int\ f}\}$ (T-VC)
  while x < y^2 {          // $z_1 \sqsupseteq z_0 \sqcup z_2, \mathtt{int} \sqsupseteq x_0$,
                           // $\mathtt{int} \sqsupseteq y_0$ (T-WHILE)
    z.f = z^3              // $z_2 \sqsupseteq z_1[f \mapsto z_1]$ (T-FV)
} }
```

Eliminating for each of the constraint variables contained in the above yields the following constraint sets (left) and extracted variable typings (right):

$$\begin{array}{ll} \mathcal{C}_{\{\$\}} = \{\mathtt{void} \sqsupseteq \$\} & \implies \Sigma(\$) = \mathtt{void} \\ \mathcal{C}_{\{\mathtt{x}_0\}} = \{\mathtt{x}_0 \sqsupseteq \mathtt{int}\} & \implies \Sigma(\mathtt{x}_0) = \mathtt{int} \\ \mathcal{C}_{\{\mathtt{y}_0\}} = \{\mathtt{y}_0 \sqsupseteq \mathtt{int}\} & \implies \Sigma(\mathtt{y}_0) = \mathtt{int} \\ \mathcal{C}_{\{\mathtt{z}_0\}} = \{\mathtt{z}_0 \sqsupseteq \{\mathtt{int\ f}\}\} & \implies \Sigma(\mathtt{z}_0) = \{\mathtt{int\ f}\} \\ \mathcal{C}_{\{\mathtt{z}_1\}} = \{\mathtt{z}_1 \sqsupseteq \{\mathtt{int\ f}\} \sqcup \mathtt{z}_1[\mathtt{f} \mapsto \mathtt{z}_1]\} & \\ & \implies \Sigma(\mathtt{z}_1) = \mu\mathtt{X}.(\{(\mathtt{int} \vee \mathtt{X})\ \mathtt{f}\}) \\ \mathcal{C}_{\{\mathtt{z}_2\}} = \{\mathtt{z}_2 \sqsupseteq (\{\mathtt{int\ f}\} \sqcup \mathtt{z}_2)[\mathtt{f} \mapsto \{\mathtt{int\ f}\} \sqcup \mathtt{z}_2]\} & \\ & \implies \Sigma(\mathtt{z}_2) = \mu\mathtt{X}.(\{\{\mathtt{int\ f}\} \vee \mathtt{X\ f}\}) \end{array}$$

An interesting observation lies in the difference between the type of $\mathtt{z}_1$ and $\mathtt{z}_2$. The "smallest" type contained in $\mathtt{z}_1$ is $\{\mathtt{int\ f}\}$, whilst for $\mathtt{z}_2$ it is $\{\{\mathtt{int\ f}\}\ \mathtt{f}\}$. These types correspond to the first iteration of the loop, with the latter representing the case where $\{\mathtt{int\ f}\}$ (i.e. z's initial value) was already assigned into field f of variable z. Furthermore, it is relatively easy to show that $\Sigma$ (as shown above) is a *valid* typing (under Definition 3) for the constraints generated for loopy().

The variable elimination process is trivially guaranteed to terminate. However, an important property is to show that it *preserves* solutions. That is, if a solution for the original constraint set exists, then a solution still exists a after variable elimination:

LEMMA 1 (SAFE SUBSTITUTION). *Assume* $\mathtt{e}_1, \mathtt{e}_2, \mathtt{n}_\ell, \mathcal{E}$ *and* $\Sigma$ *where* $\mathcal{E}(\Sigma, \mathtt{e}_1) \leq \Sigma(\mathtt{n}_\ell)$ *and* $\mathcal{E}(\Sigma, \mathtt{e}_2)$ *is well-defined. Then, it follows that* $\mathcal{E}(\Sigma, \mathtt{e}_2[\![\mathtt{n}_\ell \mapsto \mathtt{e}_1]\!]) \leq \mathcal{E}(\Sigma, \mathtt{e}_2)$.

PROOF. Proof omitted for brevity — see [24] for details. □

THEOREM 1 (ELIMINATION PRESERVATION). *Let* $\mathcal{C}_\mathcal{X}$ *be a constraint set in single assignment form where* $\{\mathtt{n}_\ell \sqsupseteq \mathtt{e}\} \subseteq \mathcal{C}_\mathcal{X}$, *and* $\Sigma$ *an arbitrary typing. If* $\Sigma \models \mathcal{C}_\mathcal{X}$ *then,* $\Sigma \models \mathcal{C}_{\mathcal{X}-\{\mathtt{n}_\ell\}}$ *for any* $\mathtt{n}_\ell \in \mathcal{X}$.

PROOF. Proof omitted for brevity — see [24] for details. □

## 4.3 Type Extraction

Given the final constraint set $\mathcal{C}_{\{\mathtt{n}_\ell\}}$ for a variable $\mathtt{n}_\ell$, the remaining challenge is to extract a type for $\mathtt{n}_\ell$. In such case, we know there is a single constraint of the form $\mathtt{n}_\ell \sqsupseteq \mathtt{e} \in \mathcal{C}_{\{\mathtt{n}_\ell\}}$ where $\mathtt{e}$ either uses no variables (i.e. it's non-recursive) or uses at most $\mathtt{n}_\ell$ (i.e. it's recursive). For the non-recursive case, this is straight-forward as $\mathcal{E}(\emptyset, \mathtt{e})$ (if it is well-defined) gives the typing for $\mathtt{n}_\ell$ (recall $\mathcal{E}(\Sigma, \mathtt{e})$ from Definition 3). For example, for $\mathtt{n}_\ell \sqsupseteq \{\mathtt{int\ f}\}[\mathtt{f} \mapsto \mathtt{any}]$ we have $\mathcal{E}(\emptyset, \{\mathtt{int\ f}\}[\mathtt{f} \mapsto \mathtt{any}]) = \{\mathtt{any\ f}\}$. If $\mathcal{E}(\emptyset, \mathtt{e})$ is not well-defined (e.g. $\mathcal{E}(\emptyset, \mathtt{int.f})$) then the original program contained a type error. For the recursive case, things are more involved. Given a recursive constraint of the form $\mathtt{n}_\ell \sqsupseteq \mathtt{e}$ (i.e. where $\mathtt{n}_\ell$ is used in $\mathtt{e}$), we first check no other $\mathtt{n}_\lambda$ is used in $\mathtt{e}$ (if not we default to rejecting the program — see §4.4), and then proceed as follows:

**Base Extraction.** To extract the base case, we use the following function, where $\bullet$ indicates a path exists to the recursive variable, $\mathtt{n}_\ell$, being extracted:

$$
\begin{aligned}
\mathcal{B}(\mathtt{n}_\ell, \mathtt{T}) &= \mathtt{T} && (1)\\
\mathcal{B}(\mathtt{n}_\ell, \mathtt{n}_\ell) &= \bullet && (2)\\
\mathcal{B}(\mathtt{n}_\ell, \mathtt{e.f}) &= \bullet \textbf{ if } \mathcal{B}(\mathtt{n}_\ell, \mathtt{e}) = \bullet && (3)\\
\mathcal{B}(\mathtt{n}_\ell, \mathtt{e.f}) &= \bigvee \mathtt{T}_\mathtt{i} \textbf{ if } \mathcal{B}(\mathtt{n}_\ell, \mathtt{e}) = \bigvee\{\ldots, \mathtt{T}_\mathtt{i}\ \mathtt{f}, \ldots\} && (4)\\
\mathcal{B}(\mathtt{n}_\ell, \mathtt{e}_1[\mathtt{f} \mapsto \mathtt{e}_2]) &= \bullet \textbf{ if } \mathcal{B}(\mathtt{n}_\ell, \mathtt{e}_1) = \bullet \textbf{ or } \mathcal{B}(\mathtt{n}_\ell, \mathtt{e}_2) = \bullet && (5)\\
\mathcal{B}(\mathtt{n}_\ell, \mathtt{e}_1[\mathtt{f} \mapsto \mathtt{e}_2]) &= \\
&\bigvee\{\overline{\mathtt{T\ f}}\}[\mathtt{f} \mapsto \mathtt{T}] \textbf{ if } \mathcal{B}(\mathtt{e}_1) = \bigvee\{\overline{\mathtt{T\ f}}\} \textbf{ and } \mathcal{B}(\mathtt{e}_2) = \mathtt{T} && (6)\\
\mathcal{B}(\mathtt{n}_\ell, \textstyle\bigsqcup \mathtt{e}_\mathtt{i}) &= \bigvee \mathtt{T}_\mathtt{j} \textbf{ forall } \mathtt{T}_\mathtt{j} \textbf{ where } \exists \mathtt{i}.\mathcal{B}(\mathtt{n}_\ell, \mathtt{e}_\mathtt{i}) = \mathtt{T}_\mathtt{j} && (7)
\end{aligned}
$$

Essentially, this factors out expressions which cannot generate concrete types (i.e. because they reference the recursive variable $\mathtt{n}_\ell$). For example, we have $\mathcal{B}(\mathtt{z}_1, \{\mathtt{int\ f}\} \sqcup \mathtt{z}_1[\mathtt{f} \mapsto \mathtt{z}_1]) = \{\mathtt{int\ f}\}$ and $\mathcal{B}(\mathtt{z}_2, (\{\mathtt{int\ f}\} \sqcup \mathtt{z}_2)[\mathtt{f} \mapsto \{\mathtt{int\ f}\} \sqcup \mathtt{z}_2]) = \{\{\mathtt{int\ f}\}\ \mathtt{f}\}$ for the recursive constraints generated for `loopy()` above.

**Base Substitution.** To extract a type for $\mathtt{n}_\ell$ we exploit knowledge of the $\mathtt{e}_1[\mathtt{f} \mapsto \mathtt{e}_2]$ construct using the following substitution function:

$$
\begin{aligned}
\mathcal{S}(\Sigma, \mathtt{T}) &= \mathtt{T} && (1)\\
\mathcal{S}(\Sigma, \mathtt{n}_\ell) &= \mathtt{T} \textbf{ if } \{\mathtt{n}_\ell \mapsto \mathtt{T}\} \subseteq \Sigma && (2)\\
\mathcal{S}(\Sigma, \mathtt{e}_1.\mathtt{f}) &= \mathtt{e}_2.\mathtt{f} \textbf{ if } \mathcal{S}(\Sigma, \mathtt{e}_1) = \mathtt{e}_2 && (3)\\
\mathcal{S}(\Sigma, \mathtt{e}_1[\mathtt{f} \mapsto \mathtt{e}_2]) &= \mathtt{e}_3[\mathtt{f} \mapsto \mathtt{e}_2] \textbf{ if } \mathcal{S}(\Sigma, \mathtt{e}_1) = \mathtt{e}_3 && (4)\\
\mathcal{S}(\Sigma, \textstyle\bigsqcup \mathtt{e}_\mathtt{i}) &= \textstyle\bigsqcup \mathtt{e}'_\mathtt{i} \textbf{ if }\\
&\mathcal{S}(\Sigma, \mathtt{e}_1) = \mathtt{e}'_1, \ldots, \mathcal{S}(\Sigma, \mathtt{e}_\mathtt{n}) = \mathtt{e}'_\mathtt{n} && (5)
\end{aligned}
$$

For $\mathtt{e}_1[\mathtt{f} \mapsto \mathtt{e}_2]$, rule (4) substitutes into $\mathtt{e}_1$ but not $\mathtt{e}_2$. For example, $\mathcal{S}(\{\mathtt{z}_1 \mapsto \{\mathtt{int\ f}\}\}, \{\mathtt{int\ f}\} \sqcup \mathtt{z}_1[\mathtt{f} \mapsto \mathtt{z}_1]) = \{\mathtt{int\ f}\} \sqcup \{\mathtt{int\ f}\}[\mathtt{f} \mapsto \mathtt{z}_1]$.

**Final Extraction.** For a recursive constraint $\mathtt{n}_\ell \sqsupseteq \mathtt{e}_1$ we extract the base type $\mathtt{T}_\mathtt{B} = \mathcal{B}(\mathtt{n}_\ell, \mathtt{e}_1)$ and substitute to give $\mathtt{e}_2 = \mathcal{S}(\{\mathtt{n}_\ell \mapsto \mathtt{T}_\mathtt{B}\}, \mathtt{e}_1)$. The type for $\mathtt{n}_\ell$ is then determined as $\mu\mathtt{X}.\mathcal{E}(\{\mathtt{n}_\ell \mapsto \mathtt{X}\}, \mathtt{e}_2)$. For example, for $\mathtt{z}_1 \sqsupseteq \{\mathtt{int\ f}\} \sqcup \mathtt{z}_1[\mathtt{f} \mapsto \mathtt{z}_1]$ we get $\mu\mathtt{X}.(\{\mathtt{int\ f}\} \vee \{\mathtt{X\ f}\})$ and, likewise, for $\mathtt{z}_2 \sqsupseteq (\{\mathtt{int\ f}\} \sqcup \mathtt{z}_2)[\mathtt{f} \mapsto \{\mathtt{int\ f}\} \sqcup \mathtt{z}_2]$ we obtain $\mu\mathtt{X}.(\{\{\mathtt{int\ f}\} \vee \mathtt{X\ f}\} \vee \{\{\mathtt{int\ f}\} \vee \mathtt{X\ f}\})$.

## 4.4 Limitations

The typing procedure described above is not complete because it is possible (in some cases) that generated constraints contain multiple variables in the right-hand side after elimination:

```
void loopy(int x, int y) { // x₀ ⊒ int, y₀ ⊒ int,
                           // void ⊒ $ (T-FUN)
  z = {f : 1}¹             // z₀ ⊒ {int f} (T-VC)
  while x < y² {           // z₁ ⊒ z₀ ⊔ z₂, int ⊒ x₀,
                           // int ⊒ y₀ (T-WHILE)
    z.f = z³               // z₂ ⊒ z₁[f ↦ z₁] (T-FV)
  }
  while x < y⁴ {           // z₃ ⊒ z₁ ⊔ z₄, int ⊒ x₀,
                           // int ⊒ y₀ (T-WHILE)
    z.f = z⁵               // z₄ ⊒ z₃[f ↦ z₃] (T-FV)
} }
```

In this case, we have the following for $\mathtt{z}_3$:

$$
\begin{aligned}
&\mathcal{C}_{\{\mathtt{z}_1,\mathtt{z}_3\}} = \{\mathtt{z}_1 \sqsupseteq \{\mathtt{int\ f}\} \sqcup \mathtt{z}_1[\mathtt{f} \mapsto \mathtt{z}_1], \mathtt{z}_3 \sqsupseteq \mathtt{z}_1 \sqcup \mathtt{z}_3[\mathtt{f} \mapsto \mathtt{z}_3]\}\\
&\hookrightarrow \mathcal{C}_{\{\mathtt{z}_3\}} = \{\mathtt{z}_3 \sqsupseteq \{\mathtt{int\ f}\} \sqcup \mathtt{z}_1[\mathtt{f} \mapsto \mathtt{z}_1] \sqcup \mathtt{z}_3[\mathtt{f} \mapsto \mathtt{z}_3]\}
\end{aligned}
$$

Here, we have not successfully eliminated $\mathtt{z}_1$ from $\mathcal{C}_{\{\mathtt{z}_3\}}$ *because it was a recursive constraint*. Therefore, in some cases, our extraction procedure cannot be applied and we must reject the program (even if it could, in principle, be typed). A more expressive language of constraints would help overcome this limitation.

**Claim.** Our typing procedure can be used to type many interesting examples (such as `loopy()` from above). Furthermore, it is trivial to show that it is both sound and complete for sets of non-recursive constraints. Thus, our procedure is at least as good as the dataflow-based approach outlined in §3 with the added benefit of guaranteed termination. Observe that we need not be concerned about whether our extraction procedure is sound or not. This is because we can simply extract a typing and then *certify* via Definition 3 that it does (or does not) satisfy the generated constraints. And, of course, if it does not satisfy the constraints we reject the program (for safety).

## 5. RELATED WORK

Numerous systems have been developed for object-oriented languages (e.g. [22, 20, 2, 29, 6, 9]). These, almost exclusively, assume the original program is completely untyped and employ set constraints (see [1, 13]) as the mechanism for inferring types. As such, they address a somewhat different problem to that studied here. To perform type inference, such systems generate constraints from the program text, formulate them as a directed graph and solve them using an algorithm similar to transitive closure.

Our earlier work on flow-typing [25] considers the problem of handling type tests in a sound and complete manner. The aim is to automatically retype variables as a result of runtime type tests. Consider a variable `x` which has type $\mathtt{T}_1$ and is the subject of a type test, such as `x instanceof` $\mathtt{T}_2$. It should follow that variable `x` automatically has type $\mathtt{T}_1 \wedge \mathtt{T}_2$ on the true branch (i.e. the intersection of its original type and the tested type). The key challenge is that, on the false branch, it should have type $\mathtt{T}_1 \wedge \neg\mathtt{T}_2$ (i.e. the intersection of its original type and everything *except* the tested type). Developing a type system which supports union, intersection and negation types which is both sound and complete is a significant algorithmic challenge, and our solution relies on a carefully constructed normal form representation of types. Note that the system presented in [25] differs from that presented here, as it does not support recursive types at all and, hence, termination is not a consideration.

Palsberg and O'Keefe consider the problem of finding a type system equivalent to a constraint-based safety analysis [21]. They find that a type system previously studied by Amadio and Cardelli (which includes subtyping and recursive types [4]) accepts exactly the same set of programs as the particular safety analysis they examined. Their work shows some similarity with the problem studied in this paper. In particular, Palsberg and O'Keefe develop a constraint-based type inference where typings are generated by solving constraints and extracting a least solution for each variable. However, their type system does not include union types and this limits the possible constraint forms needing to be considered. As such, the problem of extracting a typing from a constraint set is strictly simpler in their system than that studied here.

The work of Guha *et al.* focuses on flow-sensitive type checking for JavaScript [12]. The system retypes variables as a result of runtime type tests, although only simple forms are permitted. Recursive data types are not supported, although structural subtyping would be a natural fit here. Tobin-Hochstadt and Felleisen consider the problem of typing previously untyped Racket (aka Scheme) programs and develop a technique called *occurrence typing* [28]. They employ union types to increase the range of possible values from the untyped world which can be described, but do not permit retyping through assignment. The earlier work of Aiken *et al.* is similar to that of Tobin-Hochstadt and Felleisen [3]. This operates on a function language with single-assignment semantics. They support more expressive types, but do not consider recursive structural types. Furthermore, instead of type checking directly on the AST, conditional set constraints are generated and solved. Following the soft typing discipline, their approach is to insert runtime checks at points which cannot be shown type safe.

Finally, the Java Bytecode Verifier employs flow typing. Since locals and stack locations are untyped in Java Bytecode, it must infer their types to ensure type safety. A dataflow analysis is used to do this [16], although the problem is simpler than that studied here since one cannot retype through field assignment.

## 6. CONCLUSION

We presented a small calculus, FT, for reasoning about flow typing systems motivated from our experiences developing Whiley [15, 25]. This characterises a flow-typing problem not suitable for a dataflow-style solution, because this requires a fix-point computation over typing environments which, unfortunately, may not terminate. We then presented a novel constraint-based formulation of typing which is guaranteed to terminate. This provides a foundation for others developing such flow typing systems. More details, including proofs of progress and preservation for FT, can be found in [24]. Finally, whilst our language of constraints is similar to previous constraint-based type inference systems (e.g. [22, 2, 29, 6]), the key novelty of our approach lies in a mechanism for extracting recursive types from constraints via elimination and substitution.

**Acknowledgements.** This work is supported by the Marsden Fund, administered by the Royal Society of New Zealand.

## 7. REFERENCES

[1] A. Aiken and E. L. Wimmers. Solving systems of set constraints. In *Proceedings of LICS*, pages 329–340, 1992.

[2] A. Aiken and E. L. Wimmers. Type inclusion constraints and type inference. In *Proc. FPCA*, pages 31–41, 1993.

[3] A. S. Aiken, E. L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *Proc. POPL*, pages 163–173, 1994.

[4] R. M. Amadio and L. Cardelli. Subtyping recursive types. *ACM TOPLAS*, 15:575–631, 1993.

[5] D. Ancona, M. Ancona, A. Cuni, and N. D. Matsakis. RPython: a step towards reconciling dynamically and statically typed OO languages. In *Proc. DLS*, pages 53–64, 2007.

[6] C. Anderson, P. Giannini, and S. Drossopoulou. Towards type inference for JavaScript. In *Proc. ECOOP*, pages 428–452, 2005.

[7] R. Cytron, J. Ferrante, B. K. Rosen, M. K. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *Proc. POPL*, pages 25–35, 1989.

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

[9] M. Furr, J.-H. An, J. Foster, and M. Hicks. Static type inference for Ruby. In *Proc. SAC*, pages 1859–1866, 2009.

[10] V. Gapeyev, M. Y. Levin, and B. C. Pierce. Recursive subtyping revealed. *JFP*, 12(6):511–548, 2002.

[11] The Groovy programming language. http://groovy.codehaus.org/.

[12] A. Guha, C. Saftoiu, and S. Krishnamurthi. Typing local control and state using flow analysis. In *Proc. ESOP*, pages 256–275, 2011.

[13] N. Heintze. Set-based analysis of ML programs. In *Proc. LFP*, pages 306–317. ACM Press, 1994.

[14] S. Hunt and D. Sands. On flow-sensitive security types. In *Proc. POPL*, pages 79–90. ACM Press, 2006.

[15] D. J.Pearce and J. Noble. Implementing alanguage with flow-sensitive + structural typing on the JVM. In *Proc. BYTECODE*, 2011.

[16] X. Leroy. Java bytecode verification: algorithms and formalizations. *JAR*, 30(3/4):235–269, 2003.

[17] C. Male, D. J. Pearce, A. Potanin, and C. Dymnikov. Java bytecode verification for @NonNull types. In *Proc. CC*, pages 229–244, 2008.

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

[19] F. Nielson, H. R. Nielson, and C. L. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[20] N. Oxhøj, J. Palsberg, and M. Schwartzbach. Making type inference practical. In *Proc. ECOOP*, pages 329–349, 1992.

[21] J. Palsberg and P. O'Keefe. A type system equivalent to flow analysis. *ACM TOPLAS*, 17(4):576–599, 1995.

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

[23] D. J. Pearce. JPure: a modular purity system for Java. In *Proc. CC*, volume 6601 of *LNCS*, pages 104–123, 2011.

[24] D. J. Pearce. A calculus for constraint-based flow typing. Technical Report ECSTR12-10, Victoria University of Wellington, 2012.

[25] D. J. Pearce. Sound and complete flow typing with unions, intersections and negations. In *Proc. VMCAI*, pages 335–354, 2013.

[26] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[27] A. Russo and A. Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *Proc. CSF*, pages 186–199, 2010.

[28] S. Tobin-Hochstadt and M. Felleisen. Logical types for untyped languages. In *Proc. ICFP*, pages 117–128, 2010.

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