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.

1. Introduction

Type inference is useful for simplifying and reasoning about statically typed languages. Scala [1], C#3.0 [2], OCaml [3] and, most recently, Java 7 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 [4] or Python [5]).

Traditional type inference follows the approach of Hindley-Milner [6, 7], where exactly one type is 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 [8, 9, 10, 11, 12, 13], information flow [14, 9, 15], purity checking [16] and more [8, 11, 17, 18, 19, 20, 21]. Few languages exist which incorporate flow typing directly. Typed Racket [22, 20] 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 [23, 24, 25, 26]. Finally, Groovy 2.0 has very recently incorporated an optional flow typing system [27].

1.1 Flow Typing

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 [28] provides an excellent illustration:

```
public static float convert(int):
  iload 0    // load register 0 on stack
  i2f        // convert int to float
  fstore 0   // store float to register 0
  fload 0    // load register 0 on stack
  freturn    // return value on stack
```


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 [29]. This tracks the type of a variable at each program point, allowing it easily to handle the above example.

As another example, consider the following program written in Whiley [23, 24, 25, 26] — a language which exploits flow-typing to give the look-and-feel of a dynamically typed language:

```
define Point as {int x, int y}
define RealPoint as {real x, real y}

RealPoint normalise(Point p, int w, int h):
    p.x = ((real) p.x) / w
    p.y = ((real) p.y) / h
    return p
```

Here, the type of `p` is updated from `{int x, int y}` to `{real x,int y}` after `p.x` is assigned, and `{real x,real y}` after `p.y` is assigned. This is safe since Whiley employs *value semantics* for all data types. Thus, variable `p` is not a *reference* to a `Point` (as it would be in e.g. Java), rather it is a `Point`.

Flow typing can also retype variables after conditionals. A *non-null type system* (e.g. [11, 12, 13]) prevents variables which may hold **null** from being dereferenced. The following illustrates:

```
int cmp(String s1, @NonNull String s2) {
  if(s1 != null) {return s1.compareTo(s2);}
  else { return -1; }
}
```

The modifier `@NonNull` indicates a variable definitely cannot hold **null** and, hence, that it can be safely dereferenced. To deal with the above example, a non-null type system will retype variable `s1` to `@NonNull` on the true branch — thus allowing it to type check the subsequent dereference of `s1`.

The Whiley programming language also supports retyping through conditionals. This is achieved using the **is** operator (similar to `instanceof` in Java) as follows:

```
define Circle as {int x, int y, int r}
define Rect as {int x, int y, int w, int h}
define Shape as Circle | Rect

real area(Shape s):
    if s is Circle:
        return PI * s.r * s.r
    else:
        return s.w * s.h
```

A `Shape` is either a `Rect` or a `Circle` (which are both record types). The type test "`s is Circle`" determines whether `s` is a `Circle` or not. Unlike Java, Whiley automatically retypes `s` to have type `Circle` (resp. `Rect`) on the true (resp. false) branches of the **if** statement. There is no need to explicitly cast `s` to the appropriate `Shape` before accessing its fields.

1.2 Contributions

Existing flow typing systems are generally formulated in the style of a dataflow analysis (e.g. [29, 9, 12, 16]). 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 [23, 24, 25, 26], and present a novel constraint-based solution which is guaranteed to terminate. The result is a small calculus, called FT (for Flow Typing), which provides a foundation to help others when developing such flow typing systems. Finally, whilst our language of constraints is similar to previous constraint-based type inference systems (e.g. [30, 31, 32, 33, 34]), the key novelty of our approach lies in a mechanism for extracting recursive types from constraints via elimination and substitution.

1.3 Organisation

We first introduce the syntax and semantics of FT (§2). We then formulate typing rules in the dataflow style, and identify the termination problem (§3). Finally, we present our constraint-based typing rules and detail how these can be solved in finite time (§4).

2. Syntax, Semantics & Subtyping

We now introduce our calculus, called *FT* (for Flow-Typing), for formalising flow-typing problems. This was motivated from our work developing the Whiley programming language [23, 26, 24], and the calculus is specifically kept to a minimum to allow us to focus on the interesting problem. In this section, we introduce the syntax, semantics and subtyping rules for FT. In later sections we will present different formulations of the typing rules for FT.

2.1 Language of Types

The following gives a syntactic definition of types in FT:

$$\mathtt{T} ::= \mathtt{void} \mid \mathtt{any} \mid \mathtt{int} \mid \{\mathtt{T_1\ f_1}, \ldots, \mathtt{T_n\ f_n}\} \mid \mathtt{T_1} \vee \mathtt{T_2} \mid \mu\mathtt{X.T} \mid \mathtt{X}$$

Here, **`void`** represents the empty set of values (i.e. $\bot$), whilst **`any`** the set of all possible values (i.e. $\top$). Also, $\{\mathtt{T_1\ f_1}, \ldots, \mathtt{T_n\ f_n}\}$ 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 generally useful in flow typing systems, as they can characterise information flow at meet points in the control-flow graph. Types of the form $\mu\mathtt{X.T}$ 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 *equi-recursively* [35]. That is, recursive types and their unfoldings are not distinguished. For example, $\mu\mathtt{X}.(\mathtt{int} \vee \{\mathtt{int\ data}, \mathtt{X\ next}\})$ and $\mathtt{int} \vee \{\mathtt{int\ data}, \mu\mathtt{X}.(\mathtt{int} \vee \{\mathtt{int\ data}, \mathtt{X\ next}\})\ \mathtt{next}\}$ (i.e. it's one-step unfolding) are considered identical, and so on. Thus, we do not need explicit cases for handling recursive types as, whenever we encounter $\mu\mathtt{X.T}$, we may implicitly unfold it to $\mathtt{T}[\mathtt{X} \mapsto \mu\mathtt{X.T}]$ as necessary. Finally, recursive types are restricted to being *contractive* [36], which prohibits non-sensical types of the form $\mu\mathtt{X.X}$ and $\mu\mathtt{X}.(\mathtt{X} \vee \ldots)$.

2.2 Type Semantics

To better understand the meaning of types in FW, it is helpful to give a *semantic interpretation* (following e.g. [31, 37, 38, 39]). The aim is to give a set-theoretic model where subtype corresponds to subset. The *domain* $\mathbb{D}$ of values in our model consists of the integers and all records constructible from values in $\mathbb{D}$:

$$\mathbb{D} = \mathbb{Z} \cup \left\{ \{\mathtt{f_1} : \mathtt{v_1}, \ldots, \mathtt{f_n} : \mathtt{v_n}\} \mid \mathtt{v_1} \in \mathbb{D}, \ldots, \mathtt{v_n} \in \mathbb{D} \right\}$$

DEFINITION 1 (Type Semantics). *Every type* $\mathtt{T}$ *is characterized by the set of values it accepts, given by* $[\![\mathtt{T}]\!]$ *and defined as follows:*

$$\begin{aligned} [\![\mathtt{any}]\!] &= \mathbb{D} \\ [\![\mathtt{int}]\!] &= \mathbb{Z} \\ [\![\{\mathtt{T_1\ f_1}, \ldots, \mathtt{T_n\ f_n}\}]\!] &= \{\mathtt{f_1} : \mathtt{v_1}, \ldots, \mathtt{f_n} : \mathtt{v_n}\}, \\ &\qquad \textit{forall}\ \mathtt{v_1} \in [\![\mathtt{T_1}]\!], \ldots, \mathtt{v_n} \in [\![\mathtt{T_n}]\!] \\ [\![\mathtt{T_1} \vee \ldots \vee \mathtt{T_n}]\!] &= [\![\mathtt{T_1}]\!] \cup \ldots \cup [\![\mathtt{T_n}]\!] \end{aligned}$$

It is important to distinguish the *syntactic* representation from the *semantic* model of types. The former corresponds to a physical machine representation, whilst the latter is a mathematical ideal. As such, the syntactic representation diverges from the semantic model and, to compensate, we must establish a correlation between them. For example $\{\mathtt{int} \vee \{\mathtt{int\ x}\}\ \mathtt{f}\}$ and $\{\mathtt{int\ f}\} \vee \{\{\mathtt{int\ x}\}\ \mathtt{f}\}$ have distinct syntactic representations, but are semantically indistinguishable. For simplicity, in this paper, we assume one cannot distinguish a type from its equivalences. In practice, any algorithm for representing types would need to address this (e.g. by using canonical forms) but this is largely orthogonal to the issue at hand.

2.3 Subtyping

Amadio and Cardelli were the first to show that subtyping in the presence of recursive types was decidable [36]. Their system included function types, $\top$ and $\bot$. Kozen *et al.* improved this by developing an $\mathcal{O}(n^2)$ algorithm [40]. The system presented here essentially extends this in a straightforward manner. Gapeyev *et al.* give an excellent overview of the subject [41] and, indeed, our subtype relation is very similar to theirs.

In a nominal type system, types correspond to *trees* and, thus, the subtype operator can be defined using rules such as:

$$\frac{\mathtt{T_1} \leq \mathtt{T'_1}\ \ldots\ \mathtt{T_n} \leq \mathtt{T'_n}}{\{\mathtt{T_1\ f_1}, \ldots, \mathtt{T_n\ f_n}\} \leq \{\mathtt{T'_1\ f_1}, \ldots, \mathtt{T'_n\ f_n}\}}$$

Here, a strong property holds that the "height" of $\mathtt{T_1}$ is strictly less than e.g. $\{\mathtt{T_1\ f}\}$ — leading to a simple proof of termination since every type has finite height. In a structural type system, like FW, types correspond to graphs not trees. Defining the subtype operator using rules such as above leads to non-termination in the presence of cycles. To resolve this we employ ideas from co-induction [41]. 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 $\mathcal{C}$. To show $\mathtt{T_1}$ is a subtype of $\mathtt{T_2}$, we use the rules of Figure 1 starting with no assumptions:

DEFINITION 2 (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. As we ascend a typing derivation comparing components of $\mathtt{T_1}$ and $\mathtt{T_2}$, *the size of the assumptions set* $\mathcal{C}$ *always increases*. The S-INDUCT rule is critical here, as it protects against infinite recursion (by, essentially, treating the assumption set $\mathcal{C}$ as a "visited" set) — this follows the standard treatment of recursive types (see e.g. [35, 41]). Additionally, the size of $\mathcal{C}$ can be bounded as follows: let m (resp. n) be the number of nodes in the type graph of $\mathtt{T_1}$ (resp. $\mathtt{T_2}$); then, every addition to $\mathcal{C}$ made by a rule of Figure 1 corresponds to a pair (v, w), where v and w are (respectively) nodes in the type graph of $\mathtt{T_1}$ and $\mathtt{T_2}$ — thus, $|\mathcal{C}|$ is $O(m \cdot n)$.

Apart from assumption sets, the rules of Figure 1 are mostly straightforward. Subtyping of records is via rule S-REC which allows for *depth* but (for simplicity) not *width* [35]. Thus, it follows that $\{\mathtt{T_1\ f_1}, \ldots, \mathtt{T_n\ f_n}\} \leq \{\mathtt{T'_1\ g_1}, \ldots, \mathtt{T'_m\ g_m}\}$ if $\mathtt{n} = \mathtt{m}$ and $\forall \mathtt{1} \leq \mathtt{i} \leq \mathtt{n}.(\mathtt{f_i} = \mathtt{g_i} \wedge \mathtt{T_i} \leq \mathtt{T'_i})$ (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.

Subtyping:

$$\frac{}{\mathtt{T} \leq \mathtt{T} \lfloor \mathcal{C}} \quad \text{(S-REFLEX)}$$

$$\frac{\{\mathtt{T}_1 \leq \mathtt{T}_2\} \subseteq \mathcal{C}}{\mathtt{T}_1 \leq \mathtt{T}_2 \lfloor \mathcal{C}} \quad \text{(S-INDUCT)}$$

$$\frac{}{\mathtt{void} \leq \mathtt{T} \lfloor \mathcal{C}} \quad \text{(S-VOID)}$$

$$\frac{}{\mathtt{T} \leq \mathtt{any} \lfloor \mathcal{C}} \quad \text{(S-ANY)}$$

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

$$\frac{\begin{array}{c}\mathcal{C}_2 = \mathcal{C}_1 \cup \{\mathtt{T}_1 \leq \mathtt{T}_2 \vee \mathtt{T}_3\} \\ \exists \mathtt{i} \in \{2,3\}.\mathtt{T}_1 \leq \mathtt{T}_\mathtt{i} \lfloor \mathcal{C}_2\end{array}}{\mathtt{T}_1 \leq \mathtt{T}_2 \vee \mathtt{T}_3 \lfloor \mathcal{C}_1} \quad \text{(S-UNION1)}$$

$$\frac{\begin{array}{c}\mathcal{C}_2 = \mathcal{C}_1 \cup \{\mathtt{T}_1 \vee \mathtt{T}_2 \leq \mathtt{T}_3\} \\ \mathtt{T}_1 \leq \mathtt{T}_3 \lfloor \mathcal{C}_2 \quad \mathtt{T}_2 \leq \mathtt{T}_3 \lfloor \mathcal{C}_2\end{array}}{\mathtt{T}_1 \vee \mathtt{T}_2 \leq \mathtt{T}_3 \lfloor \mathcal{C}_1} \quad \text{(S-UNION2)}$$

$$\frac{\begin{array}{c}\mathtt{T} = \{\mathtt{T}_1\ \mathtt{f}_1 \ldots, \mathtt{T}_\mathtt{i} \vee \mathtt{T}'_\mathtt{i}\ \mathtt{f}_\mathtt{i} \ldots, \mathtt{T}_n\ \mathtt{f}_n\} \\ \mathtt{S}_1 = \{\mathtt{T}_1\ \mathtt{f}_1 \ldots, \mathtt{T}_\mathtt{i}\ \mathtt{f}_\mathtt{i}, \ldots, \mathtt{T}_n\ \mathtt{f}_n\} \\ \mathtt{S}_2 = \{\mathtt{T}_1\ \mathtt{f}_1 \ldots, \mathtt{T}'_\mathtt{i}\ \mathtt{f}_\mathtt{i}, \ldots, \mathtt{T}_n\ \mathtt{f}_n\}\end{array}}{\mathtt{T} \leq \mathtt{S}_1 \vee \mathtt{S}_2 \lfloor \mathcal{C}} \quad \text{(S-UNION3)}$$

Figure 1. Subtyping rules for FT.

Syntax:

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

Figure 2. Syntax for FT. Here, $\mathtt{n}, \mathtt{m}$ represent variable identifiers, whilst $\mathtt{i}$ represents the integer constants.

Java), but *value* types. Rule S-UNION3 is perhaps the most interesting, as it captures distributivity over records. For example, $\{\mathtt{int} \vee \{\mathtt{int}\ \mathtt{x}\}\ \mathtt{f}\} \leq \{\mathtt{int}\ \mathtt{f}\} \vee \{\{\mathtt{int}\ \mathtt{x}\}\ \mathtt{f}\}$ holds under S-UNION3.

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 $\mathtt{T}_1 \vee \mathtt{T}_2$.

2.3.1 Subtype Soundness and Completeness

We now briefly reconsider the relationship between the *syntactic* and *semantic* notions of subtyping. Recall that, in the former, subtyping is defined by the algorithmic rules given in Figure 1 whilst, in the latter, subtyping corresponds to the subset relation between the semantic sets describing a type (i.e. Definition 1).

We now state the Soundness and Complete Theorems which establish a formal connection between the semantic and syntactic notions of subtyping.

THEOREM 1 (Subtype Soundness). *Let* $\mathtt{T}$ *and* $\mathtt{T}'$ *be types where* $\mathtt{T} \leq \mathtt{T}'$. *Then,* $[\![\mathtt{T}]\!] \subseteq [\![\mathtt{T}']\!]$.

THEOREM 2 (Subtype Completeness). *Let* $\mathtt{T}$ *and* $\mathtt{T}'$ *be types where* $[\![\mathtt{T}]\!] \subseteq [\![\mathtt{T}']\!]$. *Then,* $\mathtt{T} \leq \mathtt{T}'$.

Whilst we have not given proofs of these theorems, it is relatively easy to see they hold by inspection. A formal proof of these properties, however, is quite involved and outside the scope of this report. The key challenge is that, due to the need for the assumption sets $\mathcal{C}$, a standard structural induction cannot be applied (see e.g. [42, 43, 44] for more on this).

2.4 Syntax

Figure 2 gives the syntax of FT where $[\![\cdot]\!]^\ell$ is not part of the syntax but (following [45]) identifies the distinct program points and associates each with a unique label ℓ (these will be explained later). An example FT program is given below:

```
int f(int x) {
  y = 1[1]
  z = {f : 1}[2]
  while x < y[3] { x = z.f[4] }
  return x[5]
}
```

Here, we see how each distinct program point has a unique label. 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). Furthermore, it is relatively easy to add additional constructs such as **if**–**else** statements, function invocation, arithmetic, etc.

2.5 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, Δ is the *runtime environment*, whilst $\mathtt{v}$ denotes *runtime values*. A runtime environment Δ maps variables to their current runtime value.

In Figure 3, `halt(v)` is used to indicate the machine has halted producing value $\mathtt{v}$. This must be distinguished from the notion of 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 $\mathtt{n} = \mathtt{m}.\mathtt{f}$ 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 $\mathtt{m}$ currently holds a record value containing at least field $\mathtt{f}$. Thus, in the case that $\mathtt{m}$ does not currently hold a record value, or that it holds a record value which does not contain a field $\mathtt{f}$, 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 always succeed*. This is captured in rule R-FV, where the record value being assigned is updated with a (potentially new) field `f`. The following illustrates:

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

Semantics:

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

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

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

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

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

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

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

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

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 `{int f}`; after the subsequent assignment to field `f` this becomes `{any f}`; and, finally, after the assignment to field `g` it has type `{any f, 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. Indeed, flow typing is exploited in the Whiley language [24, 26] for this reason to give the look-and-feel of a dynamically typed language.

3. Dataflow-Based Flow Typing

We now formulate the typing rules for FT as a *dataflow analysis* (see e.g. [45]). This is an intuitive and commonly used approach (e.g. [29, 9, 12, 16, 21]). 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, Γ^{ℓ}, for each program point ℓ. This gives the types of all variables immediately before the statement at ℓ. 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 Γ^1 (which represents the environment immediately *before* line 1). The following illustrates a more complex example:

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

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

$$\Gamma^4 = \{\mathtt{x}\mapsto\mathtt{int}, \mathtt{y}\mapsto\mathtt{int}\} \sqcup \{\mathtt{x}\mapsto\mathtt{int}, \mathtt{y}\mapsto\{\mathtt{int\ g}\}\} \hookrightarrow \{\mathtt{x}\mapsto\mathtt{int}, \mathtt{y}\mapsto\mathtt{int}\vee\{\mathtt{int\ g}\}\}$$

Here, $\Gamma^4(\mathtt{y}) = \mathtt{int}\vee\{\mathtt{int\ g}\}$ as an `int` value can flow from *before* the loop, whilst `{int g}` can flow from *around* the loop. When reasoning about loops, we are tacitly assuming the loop body can be executed zero or more times — even in situations, such as above, where we could be more precise. This approach is safe (but conservative) and does not require complex reasoning (e.g. with an automated theorem prover). Furthermore, it is a common assumption (e.g. Java's treatment of definite assignment and the `final` modifier [46]).

In order to define how the least upper bound on environments is determined, we must define an appropriate partial order over environments:

DEFINITION 3 (Environment Subtyping). *Let Γ^{ℓ_1} and Γ^{ℓ_2} be typing environments. Then, we say that Γ^{ℓ_1} subtypes Γ^{ℓ_2}, denoted $\Gamma^{\ell_1} \leq \Gamma^{\ell_2}$, iff $\forall \mathtt{v} \in \mathbf{dom}(\Gamma^{\ell_2}).\Gamma^{\ell_1}(\mathtt{v}) \leq \Gamma^{\ell_2}(\mathtt{v})$.*

For example, the following hold under Definition 3:

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

Since the underlying subtype relation over types forms a join semi-lattice, it follows that environment subtyping does as well (where $\bot = \emptyset$ 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 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.

The typing rules for statements describe their *effect* on the typing environment. They are judgements of the form $\Gamma \vdash \mathtt{S} : \Gamma'$ where Γ represents the environment immediately before `S`, and Γ' represents that immediately after. Thus, the effect of statement `S` is captured in the difference between Γ and Γ'. 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. Applying T-VC yields the typing environment immediately after it, namely $\Gamma^2 = \{\mathtt{x}\mapsto\mathtt{int}, \$\mapsto\mathtt{int}\}$. Finally, T-RV confirms that `x` is a subtype of the declared return type (i.e. that $\Gamma^2(\mathtt{x}) \leq \Gamma^2(\$)$ holds).

Rule T-VC exploits the fact that values have fixed types (obtained via $\vdash \mathtt{v} : \mathtt{T}$). The requirement $\Gamma(\mathtt{m}) = \{\ldots, \mathtt{T\ f}, \ldots\}$ 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

Function Typing (dataflow):

$$\frac{\{\mathtt{n}_1 \mapsto \mathtt{T}_1, \ldots, \mathtt{n}_k \mapsto \mathtt{T}_k, \$ \mapsto \mathtt{T}\} \vdash \mathtt{B} : \Gamma}{\vdash \mathtt{T}\ \mathtt{f}(\mathtt{T}_1\ \mathtt{n}_1, \ldots, \mathtt{T}_k\ \mathtt{n}_k)\ \{\mathtt{B}\}} \quad \text{(T-FUN)}$$

Block Typing (dataflow):

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

Statement Typing (dataflow):

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

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

$$\frac{\Gamma(\mathtt{m}) = \{\ldots, \mathtt{T}\ \mathtt{f}, \ldots\}}{\Gamma \vdash [\![\mathtt{n}{=}\mathtt{m.f}]\!]^{\ell} : \Gamma[\mathtt{n} \mapsto \mathtt{T}]} \quad \text{(T-VF)}$$

$$\frac{\Gamma(\mathtt{n}) = \{\mathtt{T}_1\ \mathtt{f}_1, \ldots, \mathtt{T}_n\ \mathtt{f}_n\} \quad \mathtt{T} = \Gamma(\mathtt{n})[\mathtt{f} \mapsto \Gamma(\mathtt{m})]}{\Gamma \vdash [\![\mathtt{n.f}{=}\mathtt{m}]\!]^{\ell} : \Gamma[\mathtt{n} \mapsto \mathtt{T}]} \quad \text{(T-FV)}$$

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

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

Figure 4. Dataflow-based typing rules for FT.

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`). Finally, rule T-WHILE requires a fix-point be obtained for the typing environment produced from the body. Since this is a non-trivial process, we discuss it in more detail in the following subsection.

3.2 Termination

Computing a fix-point for a dataflow analysis is normally done using an iterative computation (see e.g. [47, 48, 45]). 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:

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

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*. Unfortunately, the lattice of types in FT has infinite height — meaning such a proof strategy will not work in this case. Observe, however, *that intuitively a valid typing of the above example should exist*:

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

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* [49, 50, 45, 51, 52, 53]). 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).

The use of a widening operator is an unsatisfactory solution to this problem. Indeed, the intuitive typing given for Γ^3 above (1) still does not converge under the rules of Figure 4 and it remains unclear what additional machinery would be necessary to achieve this. In the following section, we present a novel constraint-based solution to this flow typing problem, which is guaranteed to terminate without the need for a widening operator.

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. [54, 55, 31, 56, 57, 58]. Critically, this does not require a fix-point computation and, hence, is guaranteed to terminate. Our language of type constraints is as follows:

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

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). 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 the life of each program variable may be split across multiple constraint variables (e.g. `r` is represented by $\mathtt{r}_1, \mathtt{r}_2$ and $\mathtt{r}_3$). Those familiar with *Static Single Assignment Form* [59, 60, 61] will notice a strong similarity here.

DEFINITION 4 (Typing). *A typing, Σ, 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, $\Sigma(\mathtt{e})$ is defined as follows:*

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

Rule (3) selects field `f` from a union of one or more records containing that field (e.g. $\mathcal{E}(\emptyset, (\{\mathtt{int\ f}\} \vee \{\mathtt{any\ f}\}).\mathtt{f}) = \mathtt{int} \vee \mathtt{any}$).

Function Typing (constraints):

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

Block Typing (constraints):

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

Statement Typing (constraints):

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

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

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

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

$$\frac{\Gamma(\mathtt{n}) = \kappa}{\Gamma \vdash [\![\mathtt{return}\ \mathtt{n}]\!]^\ell : \emptyset \lfloor \{\$ \sqsupseteq \mathtt{n}_\kappa\}} \quad \text{(T-RV)}$$

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

Variable Definitions:

$$\begin{array}{rl}\mathtt{defs}(\mathtt{S}\ ;\ \mathtt{B}) & = \mathtt{defs}(\mathtt{S}) \cup \mathtt{defs}(\mathtt{B}) \\ \mathtt{defs}([\![\mathtt{n} = \ldots]\!]^\ell) & = \{\mathtt{n}\} \\ \mathtt{defs}([\![\mathtt{n}.\mathtt{f} = \ldots]\!]^\ell) & = \{\mathtt{n}\} \\ \mathtt{defs}([\![\mathtt{return}\ \mathtt{n}]\!]^\ell) & = \emptyset \\ \mathtt{defs}(\mathtt{while}\ [\![\mathtt{n} < \mathtt{m}]\!]^\ell\ \{\mathtt{B}\}) & = \mathtt{defs}(\mathtt{B})\end{array}$$

Figure 5. Constraint-Based Typing rules for FT.

Likewise, rule (4) updates the type of field `f` across a union of one or more records. Here, $\bigvee\{\overline{\mathtt{T}\ \mathtt{f}}\}$ is a short-hand notation for a union of records $\{\mathtt{T}_1^1\ \mathtt{f}_1^1, \ldots, \mathtt{T}_n^1\ \mathtt{f}_n^1\} \vee \ldots \vee \{\mathtt{T}_1^k\ \mathtt{f}_1^k, \ldots, \mathtt{T}_m^k\ \mathtt{f}_m^k\}$, 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, $\mathcal{E}(\emptyset, (\{\mathtt{int\ f}\} \vee \{\mathtt{int\ g}\})[\mathtt{f} \mapsto \mathtt{any}]) = \{\mathtt{any\ f}\} \vee \{\mathtt{any\ f}, \mathtt{int\ g}\}$.

Finally, a given FT program is considered *type safe* if a valid typing exists which satisfies all the generated typing constraints by Definition 4.

4.1 Constraint-Based Typing Rules

Figure 5 gives the constraint-based typing rules for FT which have a general form of $\Gamma_0 \vdash \mathtt{S} : \Gamma_1 \lfloor \mathcal{C}$ (except T-FUN, which is similar). Here, Γ_0 represents the typing environment immediately before `S`, whilst Γ_1 represents that immediately after. In the constraint-based formulation, a typing environment Γ 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. Γ) but with `n` mapped to ℓ. Rule T-VV constrains the type of the assigned variable to that of the right-hand side. Here, $\Gamma(\mathtt{m}) = \kappa$ determines the program point (κ) where the type variable currently representing `m` was defined ($\mathtt{m}_\kappa$).

Rule T-VF is similar to T-VC, but instead 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, rule T-WHILE is the most involved. In the rule, 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 $\mathtt{n} \in \mathtt{defs}(\mathtt{B})$ 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 [59, 60, 61]. However, our setting is simpler as we do not have unstructured control-flow.

4.2 Variable Elimination

We now begin the process of presenting our algorithm for solving the typing constraints generated for a given function. Our purpose is not to present an *efficient* algorithm, but rather one which is easy to understand and formalise.

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 the typing for $\mathtt{n}_\ell$. We begin with some formalities:

DEFINITION 5 (Variable Scoping). *Let $\mathcal{C}_\mathcal{X}$ denote a constraint set where $\mathcal{X}$ defines the variables permissible in any $\mathtt{e}_1 \sqsupseteq \mathtt{e}_2 \in \mathcal{C}_\mathcal{X}$.*

DEFINITION 6 (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 $\mathtt{n}_\ell \sqsupseteq \mathtt{e}$.*

Observe that any constraint set $\mathcal{C}_\mathcal{X}$ generated from the rules of Figure 5 is *almost* in single assignment form. That's because, by construction, 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 7 (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:* $\mathcal{C}_{\mathcal{X}-\{\mathtt{n}_\ell\}} = \{\mathtt{e}_1 \sqsupseteq \mathtt{e}_2[\![\mathtt{n}_\ell \mapsto \mathtt{e}]\!] \mid \mathtt{e}_1 \sqsupseteq \mathtt{e}_2 \in \mathcal{C}_\mathcal{X} \wedge \mathtt{e}_1 \neq \mathtt{n}_\ell\}$.

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 $\mathtt{T}$ (i.e. not an arbitrary expression). Furthermore, $\mathtt{e}_2[\![\mathtt{n}_\ell \mapsto \mathtt{e}]\!]$ substitutes all occurrences of $\mathtt{n}_\ell$ with $\mathtt{e}$ 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 the example from §3.2 which caused non-termination for an iterative fix-point solver of the dataflow typing rules (i.e. without widening):

```
void loopy(int x, int y) { // x0 ⊒ int, y0 ⊒ int,
                           // void ⊒ $ (T-FUN)
  z = {f : 1}[1]           // z0 ⊒ {int f} (T-VC)
  while x < y[2] {         // z1 ⊒ z0 ⊔ z2, int ⊒ x0,
                           // int ⊒ y0 (T-WHILE)
    z.f = z[3]             // z2 ⊒ z1[f ↦ z1] (T-FV)
} }
```

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

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

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. $\mathtt{z}$'s initial value) was already assigned into field $\mathtt{f}$ of variable $\mathtt{z}$. Furthermore, it is relatively easy to show that Σ (as shown above) is a *valid* typing (under Definition 4) 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* Σ *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 1. *By structural induction on* $\mathtt{e}_2$*, where the induction hypothesis states that the Lemma holds for any substructure of* $\mathtt{e}_2$*. Proof omitted for brevity — see [62] for details.*

THEOREM 3 (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* Σ *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 2. *Proof omitted for brevity — see [62] 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 4). 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:

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

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:

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

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. For example, it is possible (in some unusual cases) that generated constraints contain multiple variables in the right-hand side after the elimination procedure. The following illustrates such a program:

```
void loopy(int x, int y) { // x0 ⊒ int, y0 ⊒ int,
                           // void ⊒ $ (T-FUN)
  z = {f : 1}[1]           // z0 ⊒ {int f} (T-VC)
  while x < y[2] {         // z1 ⊒ z0 ⊔ z2, int ⊒ x0,
                           // int ⊒ y0 (T-WHILE)
    z.f = z[3]             // z2 ⊒ z1[f ↦ z1] (T-FV)
  }
  while x < y[2] {         // z3 ⊒ z1 ⊔ z4, int ⊒ x0,
                           // int ⊒ y0 (T-WHILE)
    z.f = z[3]             // z4 ⊒ z3[f ↦ z3] (T-FV)
  }
}
```

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

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

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 4 that it does (or does not) satisfy the generated constraints. And, of course, if it does not satisfy the constraints we simply reject the program (for safety).

4.5 Soundness

In this section, we prove two standard properties for FT, namely: *progress* and *preservation*. Roughly speaking, this corresponds to showing that a well-typed program will not get stuck during execution, and that executing one step of a well-typed program preserves the validity of typing. The following notion of a *safe abstraction* captures the relationship between type environments and their corresponding runtime environments:

DEFINITION 8 (Safe Abstraction). *Let* (Σ, Γ) *be a typing and environment and* Δ *a runtime environment. Then,* (Σ, Γ) *safely abstracts* Δ*, denoted* $(\Sigma, \Gamma) \approx \Delta$*, iff* $\boldsymbol{dom}(\Gamma) \subseteq \boldsymbol{dom}(\Delta)$ *and, for all* $\mathtt{n} \mapsto \ell \in \Gamma$*, it holds that* $\Sigma(\mathtt{n}_\ell) \models \Delta(\mathtt{n})$.

Observe that we cannot require $\mathbf{dom}(\Gamma) = \mathbf{dom}(\Delta)$, as might be expected, since runtime environments are the product of actual execution paths. Consider a **`while`** statement with a variable `n` defined in only in the body. After the statement, $\mathtt{n} \notin \Gamma$ since `n` was not defined before the loop. However, if execution had proceeded through the loop body, then we would have $\mathtt{n} \in \Delta$.

THEOREM 4 (Progress). *Assume* Δ, Σ *and* Γ *where* $(\Sigma, \Gamma) \approx \Delta$*. If* $\Gamma \vdash \mathtt{S} : \Gamma' \mid \mathcal{C}$ *and* $\Sigma \models \mathcal{C}$*, then either* $\langle \Delta, \mathtt{S\ B} \rangle \longrightarrow \langle \Delta', \mathtt{B}' \rangle$ *or* $\langle \Delta, \mathtt{S\ B} \rangle \longrightarrow \mathtt{halt(v)}$.

PROOF 3. *By case analysis on* S *over the different statement forms from Figure 2. Proof omitted for brevity — see [62] for details.*

THEOREM 5 (Preservation). *Assume* Δ, Σ, Γ *where* $(\Sigma, \Gamma) \approx \Delta$ *holds. If* $\Gamma \vdash \mathtt{S} : \Gamma' \mid \mathcal{C}$*,* $\Sigma \models \mathcal{C}$ *and* $\langle \Delta, \mathtt{S\ B} \rangle \longrightarrow \langle \Delta', \mathtt{B}' \rangle$*, then* $(\Sigma, \Gamma') \approx \Delta'$.

PROOF 4. *By case analysis on* S *over the different statement forms from Figure 2. Proof omitted for brevity — see [62] for details.*

5. Extensions

We now provide some additional discussion of our constraint-based formulation of FT and highlight a number of ways in which it could be extended.

5.1 Effective Records

One of the less intuitive aspects of our definition of a typing (i.e. Definition 4) is the support for unions of records. Henceforth, we refer to these as *effective records*.

To illustrate the value of effective records, consider the following:

```
int f(int x, int y) {
  z = {f:1, g:2}[1]
  while x < y[2] {
     z = {f:3, h:4}[3]
  }
  return z.f[4]
}
```

At the **`return`** statement, it follows that variable `z` has type $\{\mathtt{int\ f}, \mathtt{int\ g}\} \vee \{\mathtt{int\ f}, \mathtt{int\ h}\}$. Therefore, one would expect `z.f` to be type safe, given that both options have the required field `f`. And, indeed, this is a valid FT program under Definition 4 and Figure 4.

5.2 Arrays

Extending FT to support array types is fairly straightforward. Suppose we extend our language of types to include a type `[T]`, which represents an array of zero or more elements of type `T`. The following illustrates how this might work:

```
[any] f([int] arr, any val, int n, int m) {
   while n < m[1] { arr[n] = val[2] }
   return arr[3]
}
```

Here, the type of `arr` at the return statement is `[any]`. This reflects the fact that, although `arr` had type `[int]` on entry, it may now hold one or more values of type `any`. Extending FT to support arrays, such as this, is fairly straightforward. It differs from records only in that, when an element is assigned, we cannot overwrite the element type with the assigned type (as we did for fields). This is because we cannot easily tell whether all elements of the array are overwritten with the new type.

5.3 Type Tests

As discussed in the introduction, flow typing can also be used to retype variables as a result of conditionals. The following illustrates how this might work in FT:

```
int f(any x):
     if x is {int field}[1]:
          r = x.field[2]
     else:
          r = 0[3]
     return r[4]
```

Here, variable `x` is retyped on the true branch to $\{\mathtt{int\ field}\}$. At the same time, it is retyped to $\neg\{\mathtt{int\ field}\}$ on the false branch (read as the type *not* $\{\mathtt{int\ field}\}$). Thus, extending FT to support type tests requires two additional things: an **`if`**–**`else`** statement to host the type test; and, a notion of negation types of the form $\neg\mathtt{T}$. A more detailed discussion of this problem can be found in our earlier work [25].

5.4 References

Some notion of reference type would be a useful extension to FT. However, if retyping through references is permitted, care must be taken to avoid unsoundness. For example, consider the following (where **`ref`**<`T`> represents a *reference* to a value of type `T`):

```
ref<any> f(ref<int> r, any x):
     *r = x[1];
     return r[2]
```

The above program is unsafe because callers to `f()` may retain their own copy of the reference `r`. The following illustrates such a caller:

```
int g(ref<int> r1, any x):
    r2 = f(r1,x)[1]
    return *r1[2] // unsafe
```

The declared type of `r1` suggests that dereferencing it will give an **int**. However, if we permitted the above definition of `f()`, the value referenced by `r1` would have type **any** at the **return** statement — thereby invalidating our assumption. In order to enable retyping through references, one can exploit *uniqueness* types (e.g. [63, 64, 65]). These guarantee that the value referenced is not shared with others and, hence, that one can safely update its type.

6. Related Work

The first, and most widely used type inference system was developed by Hindley [6] and later independently by Milner [7]. Since then, numerous systems have been developed for object-oriented languages (e.g. [30, 31, 32, 33, 66, 67, 34, 68, 69]). These, almost exclusively, assume the original program is completely untyped and employ set constraints (see [54, 55]) 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. When the entire program is untyped, type inference must proceed across method calls (known as *interprocedural analysis*) and this necessitates knowledge of the program's call graph (in the case of languages with dynamic dispatch, this must be approximated). Typically, a constraint graph representing the entire program is held in memory at once, making these approaches somewhat unsuited to separate compilation [30]. Such systems also share a strong relationship with constraint-based program analyses (e.g. [55, 70, 58, 71]), such as *alias* or *points-to* analysis (e.g. [72, 73, 74, 75]). Finally, the language of constraints presented in §4 is similar to that used in systems such as these. However, the key novelty of our approach lies in a mechanism for extracting recursive types from constraints via elimination and substitution.

Our earlier work on flow-typing [25] considers the problem of handling type tests in a sound and complete manner (recall, we briefly discussed this problem in §5.3). 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, there is no termination problem to be addressed.

Palsberg and O'Keefe consider the problem of finding a type system equivalent to a constraint-based safety analysis [76]. They find that a type system previously studied by Amadio and Cardelli (which includes subtyping and recursive types [36]) 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 [4]. This assumes programmer annotations are given for parameters, and operates in two phases: first, a flow analysis inserts special runtime checks; second, a standard (i.e. flow-insensitive) type checker operates on the modified AST. 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; furthermore, the system assumes sequential execution (true of JavaScript), since object fields can be retyped.

Tobin-Hochstadt and Felleisen consider the problem of typing previously untyped Racket (aka Scheme) programs and develop a technique called *occurrence typing* [20]. Their system will retype a variable within an expression dominated by a type test. Like Whiley, they employ union types to increase the range of possible values from the untyped world which can be described; however, they fall short of using full structural types for capturing arbitrary structure. Furthermore, in Racket, certain forms of aliasing are possible, and this restricts the points at which occurrence typing is applicable.

The earlier work of Aiken *et al.* is similar to that of Tobin-Hochstadt and Felleisen [77]. 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.

The Java Bytecode Verifier employs flow typing [28]. Since locals and stack laocations are untyped in Java Bytecode, it must infer their types to ensure type safety. A dataflow analysis is used to do this [29], although one issue is that the Java class hierarchy does not form a join semi-lattice. To deal with this, the bytecode verifier uses a simplified least upper bound operator which ignores interfaces altogether, instead relying on runtime checks to catch type errors (see e.g. [29]). The work of Male *et al.* extends bytecode verification to check `@NonNull` types [12]. This additionally permits variables to be retyped by conditionals such as `x != null`.

Gagnon *et al.* present a technique for converting Java Bytecode into an intermediate representation with a single static type for each variable [78]. Key to this is the ability to infer static types for the local variables and stack locations used in the bytecode. Since local variables are untyped in Java bytecode, this is not always possible as they can — and often do — have different types at different points; in such situations, a variable is split as necessary into multiple variables each with a different type.

Bierman *et al.* formalise the type inference mechanism to be included in C# 3.0, the latest version of the C# language [2]. This employs a very different technique known as *bidirectional type checking*, which was first developed for System F by Pierce and Turner [79]. This approach is suitable for C# 3.0 because variables cannot have different types at different program points.

Information Flow Analysis is the problem of tracking the flow of information, usually to restrict certain flows for security reasons. Hunt and Sands use dataflow-based flow typing for tracking information flow [9]. Their system is presented in the context of a simple While language not dissimilar to our dataflow formulation. Russo *et al.* use an extended version of this system to compare dynamic and static approaches [15]. They demonstrate that a purely dynamic

system will reject programs that are considered type-safe under the Hunt and Sands system. JFlow extends Java with statically checked flow annotations which are flow-insensitive [14]. Finally, Chugh *et al.* developed a constraint-based (flow-insensitive) information flow analysis of JavaScript [80].

7. Conclusion

We have presented a small calculus, FT, for reasoning about flow typing systems which is motivated from our experiences developing the Whiley language [23, 24, 25, 26]. This characterises a flow-typing problem which is not well-suited to being solved with a dataflow analysis. This is because the dataflow formulation 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. Finally, whilst our language of constraints is similar to previous constraint-based type inference systems (e.g. [30, 31, 32, 33, 34]), the key novelty of our approach lies in a mechanism for extracting recursive types from constraints via elimination and substitution.

In the future, we would like to extend our type extraction mechanism to cover all cases (recall from §4.4 that there are cases where extraction fails). We speculate this can be achieved by ensuring that the variable elimination procedure eagerly resolves recursive constraints when they arise.

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

References

[1] The scala programming language. http://lamp.epfl.ch/scala/.

[2] G. Bierman, E. Meijer, and M. Torgersen. Lost in translation: formalizing proposed extensions to C#. In *Proc. OOPSLA*, pages 479–498, 2007.

[3] D. Remy and J. Vouillon. Objective ML: An effective object-oriented extension to ML. *TOPS*, 4(1):27–50, 1998.

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

[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. ACM Press, 2007.

[6] J. R. Hindley. The principal type-scheme of an object in combinatory logic. *Transations of the AMS*, 146:29–60, 1969.

[7] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

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

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

[10] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Proc. POPL*, pages 232–245. ACM Press, 1993.

[11] Torbjörn Ekman and Görel Hedin. Pluggable checking and inferencing of non-null types for Java. *JOT*, 6(9):455–475, 2007.

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

[13] M. Fähndrich and K. R. M. Leino. Declaring and checking non-null types in an object-oriented language. In *Proc. OOPSLA*, 2003.

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

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

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

[17] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A theory of type qualifiers. In *Proc. PLDI*, pages 192–203. ACM Press, 1999.

[18] P. Pominville, F. Qian, R. Vallée-Rai, L. Hendren, and C. Verbrugge. A framework for optimizing Java using attributes. In *Proc. CC*, 2001.

[19] C. Andreae, J. Noble, S. Markstrum, and T. Millstein. A framework for implementing pluggable type systems. In *Proc. OOPSLA*, 2006.

[20] Sam Tobin-Hochstadt and Matthias Felleisen. Logical types for untyped languages. In *Proc. ICFP*, pages 117–128, 2010.

[21] Johnni Winther. Guarded type promotion: eliminating redundant casts in Java. In *Proc. Workshop on Formal Techniques for Java-like Programs*, pages 6:1–6:8. ACM Press, 2011.

[22] Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of typed scheme. In *Proc. POPL*, pages 395–406, 2008.

[23] The Whiley programming language, http://whiley.org.

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

[25] D. J. Pearce. Sound and complete flow typing with unions, intersections and negations. In *Proc. VMCAI*, page (to appear), 2013.

[26] D. J. Pearce and J. Noble. Structural and flow-sensitive types for Whiley. Technical Report ECSTR10-23, Victoria University of Wellington, 2010.

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

[28] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison Wesley, second edition, 1999.

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

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

[31] Alexander Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In *Proc. FPCA*, pages 31–41. ACM Press, 1993.

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

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

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

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

[36] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM TOPLAS*, 15:575–631, 1993.

[37] Flemming M. Damm. Subtyping with union types, intersection types and recursive types. volume 789 of *LNCS*, pages 687–706. 1994.

[38] Castagna and Frisch. A gentle introduction to semantic subtyping. In *Proc. ICALP*, pages 198–199, 2005.

[39] A. Frisch, G. Castagna, and V. Benzaken. Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. *JACM*, 55(4):19:1–19:64, 2008.

[40] Dexter Kozen, Jens Palsberg, and Michael I. Schwartzbach. Efficient recursive subtyping. In *Proc. POPL*, pages 419–428, 1993.

[41] Vladimir Gapeyev, Michael Y. Levin, and Benjamin C. Pierce. Recursive subtyping revealed. *JFP*, 12(6):511–548, 2002.

[42] D. Ancona and G. Lagorio. Complete coinductive subtyping for abstract compilation of object-oriented languages. In *Proc. Workshop*

on Formal Techniques for Java-like Programs, pages 1:1–1:7. ACM Press, 2010.

[43] D. Ancona and G. Lagorio. Idealized coinductive type systems for imperative object-oriented programs. *RAIRO - Theoretical Informatics and Applications*, 45(1):3–33, 2011.

[44] Davide Ancona and Elena Zucca. Corecursive featherweight java. In *Proc. Workshop on Formal Techniques for Java-like Programs*, 2012.

[45] Flemming Nielson, Hanne R. Nielson, and Chris L. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

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

[47] Susan Horwitz, Alan J. Demers, and Tim Teitelbaum. An efficient general iterative algorithm for dataflow analysis. *Acta Informatica*, 24(6):679–694, 1987.

[48] F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Proc. FMPA*, pages 128–141, 1993.

[49] Jason R. C. Patterson. Accurate static branch prediction by value range propagation. In *Proc. PLDI*, pages 67–78, 1995.

[50] Tsuneo Nakanishi, Kazuki Joe, Constantine D. Polychronopoulos, and Akira Fukuda. The modulo interval: A simple and practical representation for program analysis. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 91–96. IEEE Computer Society Press, 1999.

[51] Tsuneo Nakanishi and Akira Fukuda. Value range analysis with modulo interval arithmetic. In *Proceedings of the Workshop on Software and Compilers for Embedded Systems (SCOPES)*, 2001.

[52] Suan Hsi Yong and Susan Horwitz. Pointer-range analysis. In *Proc. SAS*, volume 3148 of *LNCS*, pages 133–148. Springer-Verlag, 2004.

[53] Su and Wagner. A class of polynomially solvable range constraints for interval analysis without widenings. *Theoretical Computer Science*, 345(1):122–138, 2005.

[54] Alexander Aiken and Edward L. Wimmers. Solving systems of set constraints. In *Proceedings of LICS*, pages 329–340, 1992.

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

[56] Alexander Aiken. Set constraints: Results, applications, and future directions. In *Proceedings of the workshop on Principles and Practice of Constraint Programming (PPCP)*, volume 874. Springer-Verlag, 1994.

[57] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. Flow-insensitive points-to analysis with term and set constraints. Technical Report CSD-97-964, University of California, Berkeley, 1997.

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

[59] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark K. Wegman, and F. Kenneth Zadeck. An efficient method of computing static single assignment form. In *Proc. POPL*, pages 25–35. ACM Press, 1989.

[60] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM TOPLAS*, 13(4):451–490, 1991.

[61] Jong-Deok Choi, Vivek Sarkar, and Edith Schonberg. Incremental computation of static single assignment form. In *Proc. CC*, pages 223–237. Springer-Verlag, 1996.

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

[63] John Boyland. Alias burying: Unique variables without destructive reads. *Software — Practice and Experience*, 31(6):533–553, 2001.

[64] J. Aldrich, V. Kostadinov, and C. Chambers. Alias Annotations for Program Understanding. In *Proc. OOPSLA*, pages 311–330, 2002.

[65] Dave Clarke and Tobias Wrigstad. External uniqueness is unique enough. In *Proc. ECOOP*, volume 2743 of *LNCS*, pages 59–67. Springer-Verlag, 2003.

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

[67] Stefan Kaes. Type inference in the presence of overloading, subtyping and recursive types. In *Proc. conference on LISP and Functional Programming*, pages 193–204. ACM Press, 1992.

[68] Jonathan Eifrig, Scott Smith, and Valery Trifonov. Sound polymorphic type inference for objects. In *Proc. OOPSLA*, pages 169–184. ACM Press, 1995.

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

[70] Manuel Fähndrich, Jeffrey S. Foster, Zhendong Su, and Alexander Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Proc. PLDI*, pages 85–96. ACM Press, 1998.

[71] Bjorn De Sutter, Frank Tip, and Julian Dolby. Customization of Java library classes using type constraints and profile information. In *Proc. ECOOP*, pages 585–610. Springer-Verlag, 2004.

[72] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. Polymorphic versus monomorphic flow-insensitive points-to analysis for C. In *Proceedings of the Static Analysis Symposium (SAS)*, pages 175–198. Springer-Verlag, 2000.

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

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

[75] Marc Berndl, Ondřej Lhoták, Fneg Qian, Laurie J. Hendren, and Navindra Umanee. Points-to analysis using BDDs. In *Proc. PLDI*, pages 196–207. ACM Press, 2003.

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

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

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

[79] B. Pierce and D. Turner. Local type inference. *ACM TOPLAS*, 22(1):1–44, 2000.

[80] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for javascript. In *Proc. PLDI*, pages 50–62, 2009.