Structural and Flow-Sensitive Types for Whiley

David J. Pearce and James Noble

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

July 2011

Abstract

Modern statically typed languages require variables to be declared with a single static type, and that subtyping relationships between used-defined types be made explicit. This contrasts with dynamically typed languages, where variables are declared implicitly, can hold values of different types at different points and have no restrictions on flow (leading to ad-hoc and implicit subtyping).

We present the flow-sensitive and structural type system used in the Whiley language. This permits variables to be declared implicitly, have multiple types within a function, and be retyped after runtime type tests. Furthermore, subtyping between user-defined types is implicit, based purely on structure. The result is a statically-typed language which, for the most part, has the look and feel of a dynamic language. The typing algorithm operates in a fashion similar to dataflow analysis. Widening must be applied to ensure termination although, surprisingly, there is no loss of precision. We formalise Whiley's type system and operational semantics, and give proofs of termination and soundness.

1 Introduction

Statically typed programming languages lead to programs which are more efficient and where errors are easier to detect ahead-of-time [1, 2]. Static typing forces some discipline on the programming process. For example, it ensures at least some documentation regarding acceptable function inputs is provided. In contrast, dynamically typed languages are more flexible in nature which helps reduce overheads and increase productivity [3, 4, 5, 6]. Indeed, recent times have seen a significant shift towards dynamically typed languages [7].

Numerous attempts have been made to bridge the gap between static and dynamic languages. Scala [8], C#3.0 [9], OCaml [10] and, most recently, Java 7 all employ local type inference (in some form) to reduce syntactic overhead. Techniques such as gradual typing [11, 12], soft typing [1, 13] and hybrid typing [14] enable a transitory position where some parts of a program are statically typed, and others are not. Alternatively, global type inference can be used (in some situations) to reconstruct types "after the fact" for programs written in dynamic languages [15, 16].

Such approaches focus on working around static type systems, rather than rethinking the way static typing is done. The prevailing view remains that each variable in a static type system must have exactly one declared type and that *nominal* — rather than *structural* — typing is preferred. And yet, these choices go against the notion of a dynamic language, where variables can be assigned arbitrarily and can flow unimpeded by cumbersome, rigid type hierarchies.

We present the type system used in Whiley, a statically-typed language targeting the JVM. Whiley takes a novel approach to typing, which we refer to as *flow-sensitive typing*. This comes close to giving the flexibility of a dynamic language, with the guarantees of a statically typed language. More specifically, flow-sensitive typing offers improved error handling, reduced syntactic overhead and greater opportunity for code reuse. The technique is adopted from flow-sensitive program analysis (e.g. [17, 18, 19]), and allows variables to have different types at different points and be declared implicitly, based on the type of assigned expressions. Furthermore, unlike the majority of statically typed languages, Whiley employs structural — rather than nominal — typing. This frees programmers from the burden of developing static type hierarchies up front, allowing them to expose subtyping relationships retroactively.

Finally, an open source implementation of Whiley is freely available from `http://whiley.org` and details Whiley's JVM implementation can be found here [20].

1.1 Flow-Sensitive Types

The following demonstrates Whiley's flow-sensitive types:

```
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`" (similar to an `instanceof` test in Java) 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.

Statically typed languages typically require variables be explicitly declared. Compared with dynamically typed languages, this is an extra burden for the programmer, particularly when a variable's type can be inferred from assigned expression(s). In Whiley, local variables are never explicitly declared, rather they are declared by assignment:

```
int average([int] items):
   v = 0
   for i in items:
       v = v + items[i]
   return v / |items|
```

Here, `items` is a list of **`int`**s, whilst `|items|` returns its length. Variable `v` accumulates the sum of all elements in the list, and is declared by the assignment "`v = 0`". Since `0` has type **`int`**, `v` has type **`int`** after the assignment.

1.2 Structural Subtyping

Statically typed languages, such as Java, employ nominal typing for recursive data types. This results in rigid hierarchies which are often difficult to extend [21]. In contrast, Whiley employs *structural subtyping* of records [22] to give greater flexibility. For example, the following defines a `Border` record:

```
define Border as {int x,int y,int w,int h}
```

Any instance of `Border` has identical structure to an instance of `Rect`. Wherever a `Border` is required, a `Rect` can be provided and vice-versa — even if the `Border` definition was written long after the `Rect`, and even though no explicit connection is made between `Rect` and `Border`.

The focus on structural, rather than nominal, types in Whiley is also evident in the way instances are created:

```
bool contains(int x, int y, Border b):
    ....

bool example(int x, int y):
    b = {x: 1, y: 2, w: 10, h: 3}
    return contains(x,y,b)
```

Here, function `example()` creates a record instance with fields `x`, `y`, `w` and `h`, and assigns it to variable `b`. Despite not being associated with a name, such as `Border` or `Rect`, it can be freely passed into functions expecting such types, since they have identical *structure*.

2 Language Overview

Figure 1 provides a simple implementation of expressions, along with code for evaluating them. The types `Expr` and `Value` are algebraic data types, with the latter defining the set of allowed values. Type `Op` is an enumeration, whilst `BinOp` and `ListAccess` are records which form part of `Expr`. Parameter `env` is a map from variables to `Values`. Finally, **`null`** is used as an error condition to indicate a "stuck" state (i.e. the evaluation cannot proceed).

The code in Figure 1 makes extensive use of runtime type tests to distinguish different expression forms (e.g. "`e is int`"). These work in a similar fashion to Java's `instanceof` operator, with one important

```
define Var as string
define Op as { ADD, SUB, MUL, DIV }
define BinOp as { Op op, Expr lhs, Expr rhs }
define ListAccess as { Expr lhs, Expr rhs }

define Value as int | [Value] | null

define Expr as int | Var | BinOp | [Expr] | ListAccess

Value eval(Expr e, {Var→Value} env):
   if e is int:
       return e
   else if e is Var && e in env:
       // look up variable's value
       return env[e]
   else if e is BinOp:
       // evaluate left and right expressions
       lhs = eval(e.lhs,env)
       rhs = eval(e.rhs,env)
       // sanity check
       if !(lhs is int && rhs is int):
           return null // stuck
       // evaluate result
       switch e.op:
           case ADD:
               return lhs + rhs
           case SUB:
               return lhs - rhs
           case MUL:
               return lhs * rhs
           case DIV:
               if rhs != 0:
                   return lhs / rhs
   else if e is ListAccess:
       // evaluate src and index expressions
       src = eval(e.lhs,env)
       index = eval(e.rhs,env)
       // santity check
       if src is [Value] && index is int
          && index >= 0 && index < |src|:
           return src[index]
   else if e is [Expr]:
       lv = []
       // evaluate items in list constructor
       for i in e:
           v = eval(i,env)
           if v == null:
               return v
           else:
               lv = lv + [v]
       return lv
   // some kind of error occurred, so propagate upwards
   return null
```

Figure 1: Whiley code for a simple expression tree and evaluation function. This makes extensive use of type tests, both for distinguishing expressions and error handling. Flow-sensitive typing greatly simplifies the code, which would otherwise require numerous unnecessary casts.

difference: they operate in a flow-sensitive fashion and automatically *retype* variables after the test. As an example, consider the type test "`e is int`" on Line 11. On the true branch, variable `e` is automatically retyped to have type **`int`**. Likewise, on the false branch, `e` is now known *not* to have type **`int`** (and any attempt to retest this yields a compile-time error).

Figure 1 also employs runtime type tests to identify and propagate errors. For example, having evaluated the left- and right-hand sides of a `BinOp`, we check on Line 21 that both are **`int`** values (i.e. not list values or **`null`**). After the check, Whiley's flow-sensitive type system automatically retypes both `lhs` and `rhs` to **`int`**. For `ListAccess` expressions, we check on Line 39 that `src` is a list value, and that `index` is an **`int`**. The latter is achieved with "`index is int`". As `src` is retyped within the condition itself, the subsequent use of `|src|` on Line 40 is type safe.

Implementing our expression language in a statically-typed language, such as Java, would require code that was more cumbersome, and more verbose than that of Figure 1. One reason for this is that, in languages like Java, variables must be *explicitly* retyped after `instanceof` tests. That is, we must insert casts to update the types of tested variables and, since variables can have only one type in Java, introduce temporary variables to hold these new types. For example, after a test "`e instanceof BinOp`" we must introduce a new variable, say `r`, with type `BinOp` and assign `e` to `r` using an appropriate cast. A Java implementation would also (most likely) break up the test on Line 39, since it would otherwise need two identical casts (one inside the condition for `|src|`, and one on the true branch for `src[index]`).

In an object-oriented language, such as Java, a direct conversion of Figure 1 might not be optimal. Instead, the *visitor pattern* [23] can be used to distinguish different expression forms. Using the visitor pattern reduces the amount of explicit retyping required. This is because the different expression forms are explicitly given as parameters to the visitor methods. However, using the visitor pattern is a heavyweight solution which is not suitable in all situations. In particular, it would not eliminate all forms of explicit retyping from Figure 1. In this case, explicit variable retyping will still be required to properly handle the different values returned from `eval()`. For example, to check `BinOp`s are evaluated on **`int`** operands (Line 21), and that `src` gives a list and `index` an **`int`** (Line 39).

2.1 Value Semantics

In Whiley, all compound structures (e.g. lists, sets, and records) have *value semantics*. This means they are passed and returned by-value (as in Pascal, MATLAB or most functional languages). But, unlike functional languages (and like Pascal), values of compound types can be updated in place.

Value semantics implies that updates to a variable only affects that variable, and that information can only flow out of a function through its return value. Whiley has no general, mutable heap comparable to those found in object-oriented languages. Consider:

```
int f([int] xs):
    ys = xs
    xs[0] = 1
    ...
```

The semantics of Whiley dictate that, having assigned `xs` to `ys` as above, the subsequent update to `xs` does not affect `ys`. Arguments are also passed by value, hence `xs` is updated inside `f()` and this does not affect `f`'s caller. That is, `xs` is not a *reference* to a list of **`int`**; rather, it *is* a list of **`int`**s and assignments to it do not affect state visible outside of `f()`.

Whilst this approach may seem inefficient, a variety of techniques exist (e.g. reference counting) to ensure efficiency (see e.g. [24, 25, 26]). Indeed, the underlying implementation does pass compound structures by reference and copies them only when absolutely necessary.

2.1.1 Structural Updates.

In a dynamic language, lists and records can be updated at will. However, static type systems normally require updates to respect the element or field type in question. For example, assigning a `float` to an element of an **`int`** array is not permitted in Java. To work around this, programmers typically either clone the structure in question, or "break" the type system using casting (or similar).

Updates to list elements and record fields are always permitted in Whiley. For example:

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

[int|Point] insert(int i, Point p, [int] xs):
    xs[i] = p
    return p
```

The type of `xs` is updated to `[int|Point]` after `xs` is assigned as it now contains a `Point`.

The ability to update the types of records and lists is possible because compound structures (e.g. lists, sets, records, etc) have value semantics in Whiley. Such updates would be unsafe if aliasing were permitted as, for example, callers of `insert()` might hold aliases to `xs` and would expect its elements to be **`int`**s.

2.2 Structural Types

Statically typed languages, such as Java, employ nominal typing for recursive data types. This results in rigid hierarchies which are often difficult to extend [21]. In contrast, Whiley employs *structural subtyping* of records [22] to give greater flexibility.

Suppose we wish to extend our expressions from Figure 1 with assignment statements. A common issue arises as the left-hand side of an assignment is a restricted form of expression, often called an *lval*. In a language like Java, we can capture this nicely using interfaces:

```
interface Expr { ... }
interface LVal { ... }
class ListAccess implements Expr,LVal { ...}
class Var implements Expr,LVal { ... }
class Int implements Expr { ... }
```

However, suppose the code for expressions was part of an existing library, and we are trying to add statements after the fact. In a language like Java, this presents a problem as we cannot retroactively insert the necessary `LVal` interface to `Var` and `ListAccess`.

In Whiley, adding the notion of an `LVal` is easy to do retroactively because of structural subtyping:

```
define LVal as Var | ListAccess
define Assign as {LVal lhs, Expr rhs}

Expr parseExpression():
    ...

null|Assign parseAssign():
    le = parseExpression()
    match(":=")
    re = parseExpression()
    if le is LVal:
        return {lhs: le, rhs: re}
    else:
        return null // syntax error
```

Here, `LVal` is implicitly a subtype of `Expr` — i.e. there is no need for an explicit declaration of this, as would be required in Java. That is, they can be defined entirely separately from each other (e.g. in different files, packages or entirely separate programs) — and yet, `LVal` remains a subtype of `Expr`.

2.2.1 Structural Subtyping.

Whiley permits subtyping between recursive structural types.

```
define Link as {int data, LinkedList next}
define LinkedList as null | Link
define OrderedList as null | {
  int data, int order, OrderedList next
}

int sum(LinkedList l):
    if l is null:
        return 0
    else:
        return l.data + sum(l.next)
```

Here, we have defined a standard linked list and a specialised "ordered" list where `order < next.order` for each node (see e.g. [27]). Whiley type checks this function by showing that `OrderedList` is a structural subtype of `LinkedList` — despite this relationship not being identified explicitly in the program. Type checking in the presence of recursive structural types is a well-known and challenging problem [28, 29, 30] which is further compounded in Whiley by the presence of flow-sensitive reasoning.

2.3 Flow-Sensitive Types

We now examine Whiley's flow-sensitive type system in more detail, and identify several ways in which it leads to improved code quality.

Error Handling. Nullable references have proved a significant source of error in e.g. Java [31]. The issue is that, in such languages, one can treat *nullable* references as though they are *non-null* references [32]. Many solutions have been proposed which distinguish these two forms using static type systems (e.g. [33, 34, 35, 36, 37, 38, 39, 40]).

Whiley's flow-sensitive type system lends itself naturally to handling this problem because it supports *union types* (e.g. [41, 42]). These allow variables to hold values from different types, rather than just one type. For example:

```
null|int indexOf(string str, char c):
    ...

[string] split(string str, char c):
    idx = indexOf(str,c)
    // idx has type null|int
    if idx is int:
        // idx now has type int
        below = str[0..idx]
        above = str[idx..]
        return [below,above]
    else:
        // idx now has type null
        return [str] // no occurrence
```

Here, `indexOf()` returns the first index of a character in the string, or **`null`** if there is none. The type **`null|int`** is a union type, meaning it is either an **`int`** or **`null`**.

In the above example, Whiley's flow-sensitive type system seamlessly ensures that **`null`** is never dereferenced. This is because the type **`null|int`** cannot be treated as an **`int`**. Instead, one must first check it *is* an **`int`** using a type test, such as "`idx is int`". Whiley automatically retypes `idx` to **`int`** when this is known to be true, thereby avoiding any awkward and unnecessary syntax (e.g. a cast as required in e.g. [43, 38]).

Code Reuse. Whiley's flow-sensitive type system can expose greater opportunities for code reuse:

```
1 {string} usedVariables(Expr e):
2     if e is Var:
3         return {e}
4     else if e is BinOp || e is ListAccess:
5         l = useVariables(e.lhs)
6         r = useVariables(e.rhs)
7         return l + r   // set union
8     else if e is [Expr]:
9         ...
10    else:
11        return {}
```

On Line 5, variable `e` has type `BinOp|ListAcccess`. The use of `e.lhs` at this point is type safe, since we can perform operations common to all types of a union and, in particular, unions of records expose common fields (similar to a *common initial sequence* for `union`s of `struct`s in C [44, §6.3.2.3]).

In languages like Java, exploiting code reuse in this way requires careful planning, as common types must be explicitly related in the class hierarchy. In contrast, Whiley's flow-sensitive type system lets us exploit opportunities for code reuse in an ad-hoc fashion, as and when they occur.

3 Type System

We formalise the Whiley language using a core calculus which, in the spirit of Featherweight Java [45], we call *Featherweight Whiley (FW)*. The following gives a syntactic definition of types in FW, where overbar (e.g. $\overline{\texttt{T}}$) indicates indicates a list of items numbered consecutively (e.g. $\texttt{T}_1, \ldots, \texttt{T}_n$):

$$\texttt{T} ::= \texttt{any} \mid \texttt{void} \mid \texttt{null} \mid \texttt{int} \mid \overline{\texttt{T}} \to \texttt{T} \mid \texttt{[T]} \mid \{\, \overline{\texttt{T n}}\, \} \mid \texttt{T}_1 \vee \texttt{T}_2 \mid \mu X.\texttt{T} \mid X$$

Here, **`void`** and **`any`** represent $\perp$ and $\top$; lists are given by `[T]`; and, $\{\overline{\texttt{T n}}\}$ represents records with one or more fields. The union $\texttt{T}_1 \vee \texttt{T}_2$ is a type whose values are in $\texttt{T}_1$ *or* $\texttt{T}_2$. Types are additionally restricted to being *contractive* [29]. This prohibits types of the form $\mu\texttt{X}.\texttt{X}$ and $\mu\texttt{X}.(\texttt{X} \vee \ldots)$ and is necessary to ensure a strong connection with regular trees [32].

FW follows the *equi-recursive* approach to dealing with recursive types [32]. That is, we do not distinguish between recursive types and their unfoldings. This simplifies FW's formalism, as one does not need an explicit *unfold* operator. Any implementation of FW, however, must address the fact that two equivalent types can have distinct representations on the machine (analogous to the graph isomorphism problem). We return to discuss this in §3.4.

3.1 Semantic Interpretation

To better understand the meaning of types in FW, it is helpful to give a *semantic interpretation* (following e.g. [46, 47, 48]). The aim is to give a set-theoretic model where subtype corresponds to subset. Whilst our model ties very closely with the semantics of FW, it is simplified to avoid a circular definition between types and semantics [48]. This is often referred to as "bootstrapping" the subtype relation, and affects only function values — which we model as sets of parameter and return values.

The following defines the language of values in our model:

$$\texttt{V} ::= \texttt{null} \mid \texttt{i} \mid \texttt{[}\overline{\texttt{V}}\texttt{]} \mid \{\, \overline{\texttt{n : V}}\, \} \mid (\overline{\texttt{V}}_1, \ldots, \overline{\texttt{V}}_n) \to \overline{\texttt{V}}$$

Here, $\texttt{i} \in \mathcal{I}$ represents integer values, whilst $(\overline{\texttt{V}}_1, \ldots, \overline{\texttt{V}}_n) \to \overline{\texttt{V}}$ is our model of functions used to bootstrap the subtyping relation (note the abuse of notation here where $\overline{\texttt{V}}$ indicates a set not a list — but, this should always be clear in context). Each function value consists of sets of parameter and return values which identify those values it can accept in each parameter position, and which it may return. There is no explicit connection between individual parameter values and the corresponding return value, as this level of detail is unnecessary.

Definition 1 (Type Acceptance) *A type* T accepts *a value* V*, denoted by* $\texttt{T} \models \texttt{V}$*, defined as follows:*

$$\begin{array}{rll}
\texttt{any} & \models \texttt{V} & \\
\texttt{null} & \models \texttt{null} & \\
\texttt{int} & \models \texttt{i} & \textit{if } \texttt{i} \in \mathcal{I} \\
\texttt{[T]} & \models \texttt{[}\overline{\texttt{V}}\texttt{]} & \textit{if } \overline{\texttt{T} \models \texttt{V}} \\
\{\texttt{T}_1\ \texttt{n}_1, \ldots, \texttt{T}_n\ \texttt{n}_n\} & \models \{\texttt{n}_1 : \texttt{V}_1, \ldots, \texttt{n}_m : \texttt{V}_m\} & \textit{if } \texttt{n} < \texttt{m}, \overline{\texttt{T} \models \texttt{V}} \\
\texttt{T}_1 \vee \texttt{T}_2 & \models \texttt{V} & \textit{if } \texttt{T}_i \models \texttt{V},\ \texttt{i} \in \{1, 2\} \\
\overline{\texttt{T}} \to \texttt{T} & \models (\overline{\texttt{V}}_1, \ldots, \overline{\texttt{V}}_n) \to \overline{\texttt{V}} & \textit{if } \texttt{T} \models \overline{\texttt{V}}, \overline{\texttt{V}}_1 \models \texttt{T}, \ldots, \overline{\texttt{V}}_n \models \texttt{T}
\end{array}$$

Here, $\overline{\texttt{V}} \models \texttt{T}$ *denotes* $\neg\exists\texttt{V}.[\texttt{T} \models \texttt{V} \wedge \texttt{V} \notin \overline{\texttt{V}}]$*. In other words, every value accepted by* T *is in the set* $\overline{\texttt{V}}$*.*

Definition 2 (Type Subset) *We take* $\texttt{T}_1 \models \texttt{T}_2$ *to denote* $\forall\texttt{V}.[\texttt{T}_1 \models \texttt{V} \implies \texttt{T}_2 \models \texttt{V}]$*. In other words, that the set of values accepted by* $\texttt{T}_1$ *is a subset of those accepted by* $\texttt{T}_2$*.*

We are essentially viewing types as finite tree automatons which accept conforming trees (i.e values). The rule for accepting function values may seem somewhat strange. However, consider this example:

```
int f(any x):
    return 1

int g(int(int) fp):
    return fp(123)
```

Here, `fp` is a variable of type $\texttt{int} \to \texttt{int}$, whilst function `f()` has type $\texttt{any} \to \texttt{int}$. Intuitively, `g(&f)` should be a syntatically correct expression and, hence, $\texttt{any} \to \texttt{int}$ must subtype $\texttt{int} \to \texttt{int}$ (as is normal). Essentially, the type $\texttt{int} \to \texttt{int}$ will accept any function value which is prepared to accept any **`int`** value.

Subtyping:

$$\frac{}{\mathtt{T} \leq \mathtt{T} \downharpoonleft \mathcal{C}} \qquad \frac{\{\mathtt{T_1} \leq \mathtt{T_2}\} \subseteq \mathcal{C}}{\mathtt{T_1} \leq \mathtt{T_2} \downharpoonleft \mathcal{C}} \qquad \text{(S-REFLEX, S-INDUCT)}$$

$$\frac{}{\mathtt{void} \leq \mathtt{T} \downharpoonleft \mathcal{C}} \qquad \frac{}{\mathtt{T} \leq \mathtt{any} \downharpoonleft \mathcal{C}} \qquad \text{(S-VOID, S-ANY)}$$

$$\frac{\mathtt{T_1} \leq \mathtt{T_2} \downharpoonleft \mathcal{C} \cup \{[\mathtt{T_1}] \leq [\mathtt{T_2}]\}}{[\mathtt{T_1}] \leq [\mathtt{T_2}] \downharpoonleft \mathcal{C}} \qquad \text{(S-LIST)}$$

$$\frac{\begin{array}{c} \mathtt{n} < \mathtt{m} \\ \mathtt{T_1} = \{\mathtt{T_3\ f_3}, \ldots, \mathtt{T_m\ f_m}\} \\ \mathtt{T_2} = \{\mathtt{T'_3\ f_3}, \ldots, \mathtt{T'_n\ f_n}\} \\ \mathtt{T_3} \leq \mathtt{T'_3} \downharpoonleft \mathcal{C} \cup \{\mathtt{T_1} \leq \mathtt{T_2}\} \\ \ldots \\ \mathtt{T_n} \leq \mathtt{T'_n} \downharpoonleft \mathcal{C} \cup \{\mathtt{T_1} \leq \mathtt{T_2}\} \end{array}}{\mathtt{T_1} \leq \mathtt{T_2} \downharpoonleft \mathcal{C}} \qquad \text{(S-REC)}$$

$$\frac{\begin{array}{c} \overline{\mathtt{T}} \geq \overline{\mathtt{S}} \downharpoonleft \mathcal{C} \cup \{\overline{\mathtt{T}} \to \mathtt{T} \leq \overline{\mathtt{S}} \to \mathtt{S}\} \\ \mathtt{T} \leq \mathtt{S} \downharpoonleft \mathcal{C} \cup \{\overline{\mathtt{T}} \to \mathtt{T} \leq \overline{\mathtt{S}} \to \mathtt{S}\} \end{array}}{\overline{\mathtt{T}} \to \mathtt{T} \leq \overline{\mathtt{S}} \to \mathtt{S} \downharpoonleft \mathcal{C}} \qquad \text{(S-FUN)}$$

$$\frac{\begin{array}{c} \mathtt{i} \in \{2,3\} \\ \mathtt{T_1} \leq \mathtt{T_i} \downharpoonleft \mathcal{C} \cup \{\mathtt{T_1} \leq \mathtt{T_2} \vee \mathtt{T_3}\} \end{array}}{\mathtt{T_1} \leq \mathtt{T_2} \vee \mathtt{T_3} \downharpoonleft \mathcal{C}} \qquad \text{(S-UNION1)}$$

$$\frac{\begin{array}{c} \mathtt{T_2} \leq \mathtt{T_1} \downharpoonleft \mathcal{C} \cup \{\mathtt{T_2} \vee \mathtt{T_3} \leq \mathtt{T_1}\} \\ \mathtt{T_3} \leq \mathtt{T_1} \downharpoonleft \mathcal{C} \cup \{\mathtt{T_2} \vee \mathtt{T_3} \leq \mathtt{T_1}\} \end{array}}{\mathtt{T_2} \vee \mathtt{T_3} \leq \mathtt{T_1} \downharpoonleft \mathcal{C}} \qquad \text{(S-UNION2)}$$

$$\frac{\begin{array}{c} \mathtt{T} = \{\mathtt{T_1\ f_1} \ldots, \mathtt{T_i} \vee \mathtt{T'_i\ f_i} \ldots, \mathtt{T_n\ f_n}\} \\ \{\mathtt{T_1\ f_1} \ldots, \mathtt{T_i\ f_i} \ldots, \mathtt{T_n\ f_n}\} \leq \mathtt{S} \downharpoonleft \mathcal{C} \cup \{\mathtt{T} \leq \mathtt{S}\} \\ \{\mathtt{T_1\ f_1} \ldots, \mathtt{T'_i\ f_i} \ldots, \mathtt{T_n\ f_n}\} \leq \mathtt{S} \downharpoonleft \mathcal{C} \cup \{\mathtt{T} \leq \mathtt{S}\} \end{array}}{\mathtt{T} \leq \mathtt{S} \downharpoonleft \mathcal{C}} \qquad \text{(S-UNION3)}$$

Figure 2: Subtype rules for Featherweight Whiley. There is no need for an explicit transitivity rule.

Definition 3 (Type Equivalence) *Two types* $\mathtt{T_1}$ *and* $\mathtt{T_2}$ *are said to be* equivalent*, denoted by* $\mathtt{T_1} \equiv \mathtt{T_2}$*, if* $\mathtt{T_1} \models \mathtt{T_2}$ *and* $\mathtt{T_2} \models \mathtt{T_1}$.

Under this definition, two types are equivalent if they define the same regular tree language. Interesting cases arise when we consider *distributivity* of types. Intuitively, $\{\mathtt{int} \vee \mathtt{null\ f}\} \equiv \{\mathtt{int\ f}\} \vee \{\mathtt{null\ f}\}$. However, care must be taken as, for example $[\mathtt{int} \vee \mathtt{null}] \not\equiv [\mathtt{int}] \vee [\mathtt{null}]$ since $[\mathtt{int} \vee \mathtt{null}] \models [1, \mathtt{null}]$, but $[\mathtt{int}] \vee [\mathtt{null}] \not\models [1, \mathtt{null}]$. As another example, consider:

```
define List1 as { int|null dat, null|List1 nxt }
define List2 as { null dat, null|List2 nxt }
define List3 as { int dat, null|List3 nxt }
```

Again, $\mathtt{List1} \not\equiv \mathtt{List2} \vee \mathtt{List3}$ as, if $\mathtt{V} = \{\mathtt{dat} : \mathtt{null}, \mathtt{nxt} : \{\mathtt{dat} : 1, \mathtt{nxt} : \mathtt{null}\}\}$, then $\mathtt{List1} \models \mathtt{V}$ but $\mathtt{List2} \not\models \mathtt{V}$ and $\mathtt{List3} \not\models \mathtt{V}$.

3.2 Subtyping

Whilst the above definitions characterise types in FW, they are not constructive from an implementation perspective. As our primary goal is to develop an efficient implementation, it is important to consider such algorithmic issues. For example, consider:

```
define Link as {int dt, LinkedList nxt}
define LinkedList as null | Link
```

One possible type for `LinkedList` is $\mu\texttt{X}.(\texttt{null} \vee \{\texttt{int dt}, \texttt{X nxt}\})$, whilst another (equivalent) possibility is $\texttt{null} \vee \{\texttt{int dt}, \mu\texttt{X}.(\texttt{null} \vee \{\texttt{int dt}, \texttt{X nxt}\})\ \texttt{nxt}\}$. Under Definition 3, recursive types are equivalent to their (infinite) set of unfoldings:

$$
\begin{array}{l}
\mu X.(\texttt{null} \vee \{\texttt{int dt,X nxt}\}) \\
\equiv (\texttt{null} \vee \{\texttt{int dt}, \mu X.(\texttt{null} \vee \{\texttt{int dt,}X\ \texttt{nxt}\})\ \texttt{nxt}\}) \\
\equiv (\texttt{null} \vee \{\texttt{int dt}, (\texttt{null} \vee \{\texttt{int dt,}\mu X.(\ldots)\ \texttt{nxt}\})\ \texttt{nxt}\}) \\
\equiv \ldots
\end{array}
$$

A critical question is: *how do we implement subtyping efficiently in the presence of such types?*

Amadio and Cardelli were the first to show that subtyping in the presence of recursive types was decidable [29]. Their system included function types, $\top$ and $\bot$. Kozen *et al.* improved this by developing an $\mathcal{O}(n^2)$ algorithm [30]. The system presented here essentially extends this in a straightforward manner. Gapeyev *et al.* give an excellent overview of the subject [28] and, indeed, our subtype relation is identical to theirs, except for the inclusion of unions and other compound structures.

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

$$\frac{\texttt{T}_1 \leq \texttt{T}_2}{[\texttt{T}_1] \leq [\texttt{T}_2]}$$

Here, a strong property holds that the "height" of $\texttt{T}_1$ is strictly less than $[\texttt{T}_1]$ — 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 [28].

The subtyping rules for FW are given in Figure 2. These employ judgements of the form "$\texttt{T}_1 \leq \texttt{T}_2 \mid \mathcal{C}$", which are read as: $\texttt{T}_1$ is a subtype of $\texttt{T}_2$ under assumptions $\mathcal{C}$. Essentially, the set of assumptions $\mathcal{C}$ helps ensure the subtype operator terminates (we'll return to this shortly). Ignoring the issue of assumption sets, the rules of Figure 2 are mostly straightforward. Since lists have value semantics, $\texttt{[int]} \leq \texttt{[any]}$ holds. Subtyping of records allows for *depth* and *width* [32]. Thus, $\{\texttt{int x}, \texttt{int y}\} \leq \{\texttt{any x}\}$ by S-REC.

Rule S-UNION3 captures distributivity over records. For example, under S-UNION3, it holds that $\{\texttt{int} \vee \texttt{null x}\} \leq \{\texttt{int x}\} \vee \{\texttt{null x}\}$. The following derivation illustrates a more complex example:

$$
\begin{array}{lcl}
\texttt{T}_1 & = & \{\texttt{int} \vee \texttt{null x}, \texttt{int} \vee \texttt{null y}\} \\
\texttt{S}_1 & = & \{\texttt{int x}, \texttt{int y}\} \vee \{\texttt{null x}, \texttt{int y}\} \vee \{\texttt{int x}, \texttt{null y}\} \vee \{\texttt{null x}, \texttt{null y}\}
\end{array}
$$

---

1. $\{\texttt{int x}, \texttt{int y}\} \leq \{\texttt{int x}, \texttt{int y}\} \mid \{\ldots\}$ (S-REFLEX)
2. $\{\texttt{null x}, \texttt{int y}\} \leq \{\texttt{null x}, \texttt{int y}\} \mid \{\ldots\}$ (S-REFLEX)
3. $\{\texttt{int x}, \texttt{null y}\} \leq \{\texttt{int x}, \texttt{null y}\} \mid \{\ldots\}$ (S-REFLEX)
4. $\{\texttt{null x}, \texttt{null y}\} \leq \{\texttt{null x}, \texttt{null y}\} \mid \{\ldots\}$ (S-REFLEX)
5. $\{\texttt{int x}, \texttt{int y}\} \leq \texttt{S}_1 \mid \{\ldots\}$ (S-UNION1, 1)
6. $\{\texttt{null x}, \texttt{int y}\} \leq \texttt{S}_1 \mid \{\ldots\}$ (S-UNION1, 2)
7. $\{\texttt{int x}, \texttt{null y}\} \leq \texttt{S}_1 \mid \{\ldots\}$ (S-UNION1, 3)
8. $\{\texttt{null x}, \texttt{null y}\} \leq \texttt{S}_1 \mid \{\ldots\}$ (S-UNION1, 4)
9. $\{\texttt{int} \vee \texttt{null x}, \texttt{int y}\} \leq \texttt{S}_1 \mid \{\ldots\}$ (S-UNION3, 5+6)
10. $\{\texttt{int} \vee \texttt{null x}, \texttt{null y}\} \leq \texttt{S}_1 \mid \{\ldots\}$ (S-UNION3, 7+8)
11. $\texttt{T}_1 \leq \texttt{S}_1 \mid \emptyset$ (S-UNION3, 9+10)

Given the rules of Figure 2, we can give an algorithmic interpretation of subtyping in FW:

Definition 4 (Subtyping) *Let* $\texttt{T}_1$ *and* $\texttt{T}_2$ *be types. Then,* $\texttt{T}_1$ *is a subtype of* $\texttt{T}_2$*, denoted* $\texttt{T}_1 \leq \texttt{T}_2$*, iff* $\texttt{T}_1 \leq \texttt{T}_2 \mid \emptyset$.

To show $\texttt{T}_1$ is a subtype of $\texttt{T}_2$, we use the rules of Figure 2 starting with no assumptions. As we descend the type graph comparing components of $\texttt{T}_1$ and $\texttt{T}_2$ the set $\mathcal{C}$ *always increases*. The S-INDUCT rule is critical here, as it terminates the recursion (by, essentially, treating the assumption set $\mathcal{C}$ as a "visited" set). Furthermore, the size of $\mathcal{C}$ can be bounded as follows: let m (resp. n) be the number of nodes in the type graph of $\texttt{T}_1$ (resp. $\texttt{T}_2$); then, every addition to $\mathcal{C}$ made by a rule of Figure 2 corresponds to a pair (v, w), where v and w are (respectively) nodes in the type graph of $\texttt{T}_1$ and $\texttt{T}_2$ — thus, $|\mathcal{C}|$ is $O(m \cdot n)$. For example, consider the following type definition (where `LinkedList` is defined in §3.1):

```
define AnyList as null | {any dt, AnyList nxt}

AnyList f(LinkedList l):
   return l
```

For this function to be considered type safe, we need to show that $\mathtt{T}_1 = \mu X.(\mathtt{null}\vee\{\mathtt{int}\ \mathtt{dt}, X\ \mathtt{nxt}\})$ is a subtype of $\mathtt{S}_1 = \mu X.(\mathtt{null}\vee\{\mathtt{any}\ \mathtt{dt}, X\ \mathtt{nxt}\})$:

$$\begin{array}{lcl}
\mathtt{T}_1 & = & \mathtt{T}_3 \vee \mathtt{T}_2 \\
\mathtt{T}_2 & = & \{\mathtt{int}\ \mathtt{dt}, \mathtt{T}_1\ \mathtt{nxt}\} \\
\mathtt{T}_3 & = & \mathtt{null} \\
\mathtt{S}_1 & = & \mathtt{S}_3 \vee \mathtt{S}_2 \\
\mathtt{S}_2 & = & \{\mathtt{any}\ \mathtt{dt}, \mathtt{S}_1\ \mathtt{nxt}\} \\
\mathtt{S}_3 & = & \mathtt{null}
\end{array}$$

1.	$\mathtt{T}_3 \leq \mathtt{S}_3 \lfloor \{\mathtt{T}_1 \leq \mathtt{S}_1, \ldots\}$	(S-REFLEX)
2.	$\mathtt{int} \leq \mathtt{any} \lfloor \{\mathtt{T}_1 \leq \mathtt{S}_1, \ldots\}$	(S-ANY)
3.	$\mathtt{T}_1 \leq \mathtt{S}_1 \lfloor \{\mathtt{T}_1 \leq \mathtt{S}_1, \ldots\}$	(S-INDUCT)
4.	$\mathtt{T}_2 \leq \mathtt{S}_2 \lfloor \{\mathtt{T}_1 \leq \mathtt{S}_1, \ldots\}$	(S-REC, 2+3)
5.	$\mathtt{T}_3 \leq \mathtt{S}_1 \lfloor \{\mathtt{T}_1 \leq \mathtt{S}_1, \ldots\}$	(S-UNION1, 1)
6.	$\mathtt{T}_2 \leq \mathtt{S}_1 \lfloor \{\mathtt{T}_1 \leq \mathtt{S}_1\}$	(S-UNION1, 4)
7.	$\mathtt{T}_1 \leq \mathtt{S}_1 \lfloor \emptyset$	(S-UNION2, 5+6)

Here, the assumption $\mathtt{T}_1 \leq \mathtt{S}_1$ is propagated through the proof, and terminates the recursion at (3) with the use of S-INDUCT.

Finally, the following derivation provides a useful sanity check that $\mu\mathtt{X}.\{\mathtt{int}\vee\mathtt{null}\ \mathtt{data}, \mathtt{null}\vee\mathtt{X}\ \mathtt{next}\}$ is *not* a subtype of $\mu\mathtt{X}.\{\mathtt{int}\ \mathtt{data}, \mathtt{null}\vee\mathtt{X}\ \mathtt{next}\} \vee \mu\mathtt{X}.\{\mathtt{null}\ \mathtt{data}, \mathtt{null}\vee\mathtt{X}\ \mathtt{next}\}$:

$$\begin{array}{lcl}
\mathtt{T}_1 & = & \{\mathtt{int}\vee\mathtt{null}\ \mathtt{data}, \mathtt{null}\vee\mathtt{T}_1\ \mathtt{next}\} \\
\mathtt{S}_1 & = & \mathtt{S}_3 \vee \mathtt{S}_2 \\
\mathtt{S}_2 & = & \{\mathtt{int}\ \mathtt{data}, \mathtt{null}\vee\mathtt{S}_2\ \mathtt{next}\} \\
\mathtt{S}_3 & = & \{\mathtt{null}\ \mathtt{data}, \mathtt{null}\vee\mathtt{S}_3\ \mathtt{next}\}
\end{array}$$

1.	$\mathtt{int} \not\leq \mathtt{null} \lfloor \{\mathtt{T}_1 \leq \mathtt{S}_1, \ldots\}$	(By inspection of Fig 2)
2.	$\mathtt{null} \not\leq \mathtt{int} \lfloor \{\mathtt{T}_1 \leq \mathtt{S}_1, \ldots\}$	(By inspection of Fig 2)
3.	$\mathtt{T}_1 \not\leq \mathtt{null} \lfloor \{\mathtt{T}_1 \leq \mathtt{S}_1, \ldots\}$	(By inspection of Fig 2)
4.	$\mathtt{int} \vee \mathtt{null} \not\leq \mathtt{null} \lfloor \{\mathtt{T}_1 \leq \mathtt{S}_1, \ldots\}$	(S-UNION2, 1)
5.	$\mathtt{int} \vee \mathtt{null} \not\leq \mathtt{int} \lfloor \{\mathtt{T}_1 \leq \mathtt{S}_1, \ldots\}$	(S-UNION2, 2)
6.	$\mathtt{T}_1 \not\leq \mathtt{S}_3 \lfloor \{\mathtt{T}_1 \leq \mathtt{S}_1, \ldots\}$	(S-REC, 4)
7.	$\mathtt{T}_1 \not\leq \mathtt{S}_2 \lfloor \{\mathtt{T}_1 \leq \mathtt{S}_1, \ldots\}$	(S-REC, 5)
8.	$\mathtt{T}_1 \not\leq \mathtt{null}\vee\mathtt{S}_3 \lfloor \{\mathtt{T}_1 \leq \mathtt{S}_1, \ldots\}$	(S-UNION1, 3,6)
9.	$\mathtt{T}_1 \not\leq \mathtt{null}\vee\mathtt{S}_2 \lfloor \{\mathtt{T}_1 \leq \mathtt{S}_1, \ldots\}$	(S-UNION1, 3,7)
10.	$\mathtt{null}\vee\mathtt{T}_1 \not\leq \mathtt{null}\vee\mathtt{S}_3 \lfloor \{\mathtt{T}_1 \leq \mathtt{S}_1, \ldots\}$	(S-UNION2, 8)
11.	$\mathtt{null}\vee\mathtt{T}_1 \not\leq \mathtt{null}\vee\mathtt{S}_2 \lfloor \{\mathtt{T}_1 \leq \mathtt{S}_1, \ldots\}$	(S-UNION2, 9)
12.	$\{\mathtt{int}\ \mathtt{data}, \mathtt{null}\vee\mathtt{T}_1\ \mathtt{next}\} \not\leq \{\mathtt{null}\ \mathtt{data}, \mathtt{null}\vee\mathtt{S}_3\ \mathtt{next}\} \lfloor \{\mathtt{T}_1 \leq \mathtt{S}_1, \ldots\}$	(S-REC, 10)
13.	$\{\mathtt{int}\ \mathtt{data}, \mathtt{null}\vee\mathtt{T}_1\ \mathtt{next}\} \not\leq \{\mathtt{int}\ \mathtt{data}, \mathtt{null}\vee\mathtt{S}_2\ \mathtt{next}\} \lfloor \{\mathtt{T}_1 \leq \mathtt{S}_1, \ldots\}$	(S-REC, 11)
14.	$\{\mathtt{int}\ \mathtt{data}, \mathtt{null}\vee\mathtt{T}_1\ \mathtt{next}\} \not\leq \mathtt{S}_1 \lfloor \{\mathtt{T}_1 \leq \mathtt{S}_1\}$	(S-UNION1, 12,13)
15.	$\mathtt{T}_1 \not\leq \mathtt{S}_1 \lfloor \emptyset$	(S-UNION3, 14)

Note, in the above, S-UNION2 is used at (10) + (11); however, using S-UNION1 does not affect the outcome.

3.3 Soundness and Completeness

We now provide the necessary connections between the semantic and algorithmic interpretations. Unfortunately, performing a structural induction in the presence of assumption sets is challenging.

Definition 5 (Derivation) *For a given subtyping computation* $\mathtt{T}_1 \leq \mathtt{T}_2 \lfloor \mathcal{C}_1$*, let* $\mathcal{D}$ *denote its* derivation. *That is, the set of judgments of the form* $\mathtt{T}_3 \leq \mathtt{T}_4 \lfloor \mathcal{C}_2$ *generated by the rules of Figure 2.*

Definition 6 (Projection) *If* $\mathcal{D}$ *is a derivation, then its projection is* $\mathcal{D}^* = \{\mathtt{T}_1 \leq \mathtt{T}_2 \mid \mathtt{T}_1 \leq \mathtt{T}_2 \lfloor \mathcal{C} \in \mathcal{D}\}$.

Lemma 1 (Subtype Assumptions) *Let* $\mathtt{T}_1$ *and* $\mathtt{T}_2$ *be types where computing* $\mathtt{T}_1 \leq \mathtt{T}_2 \lfloor \emptyset$ *generates an intermediate judgment* $\mathtt{T}_3 \leq \mathtt{T}_4 \lfloor \mathcal{C}$ *which is shown to hold. If* $\mathtt{T}_1 \leq \mathtt{T}_2 \lfloor \emptyset$ *then* $\mathtt{T}_3 \leq \mathtt{T}_4 \lfloor \emptyset$.

Proof 1 *Let $\mathcal{D}_1$ be the derivation for $\mathtt{T_1} \leq \mathtt{T_2} \rfloor \emptyset$ and $\mathcal{D}_2$ the derivation for $\mathtt{T_3} \leq \mathtt{T_4} \rfloor \mathcal{C}$. By inspection of Figure 2, it follows that $\mathcal{D}_1^* \supseteq \mathcal{D}_2^*$. Let $\mathcal{C}' = \mathcal{C} - \{\mathtt{c}\}$ for some arbitrary $\mathtt{c} \in \mathcal{C}$, and $\mathcal{D}_3$ be the derivation for $\mathtt{T_3} \leq \mathtt{T_4} \rfloor \mathcal{C}'$. It follows that $\mathcal{D}_1^* \supseteq \mathcal{D}_3^* \supseteq \mathcal{D}_2^*$ as only S-INDUCT is affected by the difference $\mathcal{C}$ and $\mathcal{C}'$. We can also conclude that $\mathcal{D}_3^* = \mathtt{T_3} \leq \mathtt{T_4} \rfloor \mathcal{C}'$ was shown to hold. This is because assumptions do not interfere with the other rules from Figure 2. By an inductive argument, we can conclude $\mathtt{T_3} \leq \mathtt{T_4} \rfloor \emptyset$ is shown to hold.* □

Theorem 1 (Subtype Soundness) *Let $\mathtt{T}$ and $\mathtt{T}'$ be types where $\mathtt{T} \leq \mathtt{T}'$. Then, $\mathtt{T} \models \mathtt{T}'$.*

Proof 2 *By induction on the structure of $\mathtt{T}$ and $\mathtt{T}'$. The induction hypothesis is that if $\mathtt{T_1} \leq \mathtt{T_2} \rfloor \emptyset$ holds for some substructure of $\mathtt{T}$ and (resp.) $\mathtt{T}'$, then $\mathtt{T_1} \models \mathtt{T_2}$. Each case corresponds to a rule from Figure 2, although S-REFLEX, S-VOID, S-ANY are ignored since they follow immediately from Definition 1.*

- *Case $[\mathtt{T_1}] \leq [\mathtt{T_2}]$: By S-LIST, we have $\mathtt{T_1} \leq \mathtt{T_2} \rfloor \mathcal{C}$ and, hence, $\mathtt{T_1} \leq \mathtt{T_2} \rfloor \emptyset$ by Lemma 1. Thus, it follows from Definition 1 that $[\mathtt{T_1}] \models [\mathtt{T_2}]$.*
- *Case $\{\mathtt{T_1\ n_1}, \ldots, \mathtt{T_m\ n_m}\} \leq \{\mathtt{T_1\ T_1}, \ldots, \mathtt{T_n\ T_n}\}$ where $\mathtt{n} < \mathtt{m}$: By S-REC, we have $\mathtt{T_i} \leq \mathtt{S_i} \rfloor \mathcal{C}$ for $\mathtt{1} \leq \mathtt{i} \leq \mathtt{m}$ and, hence, $\mathtt{T_i} \leq \mathtt{S_i} \rfloor \emptyset$ by Lemma 1. This follows Definition 1 where only the first $\mathtt{n}$ fields of a record are tested for acceptance. Thus, $\{\mathtt{T_1\ n_1}, \ldots, \mathtt{T_m\ n_m}\} \models \{\mathtt{T_1\ n_1}, \ldots, \mathtt{T_n\ n_n}\}$.*
- *Case $\mathtt{T_1} \leq \mathtt{T_2} \vee \mathtt{T_3}$: By S-UNION1, we have $\mathtt{T_1} \leq \mathtt{T_2} \rfloor \mathcal{C}$ or $\mathtt{T_1} \leq \mathtt{T_3} \rfloor \mathcal{C}$ and, hence, $\mathtt{T_1} \leq \mathtt{T_2} \rfloor \emptyset$ or $\mathtt{T_1} \leq \mathtt{T_3} \rfloor \emptyset$ by Lemma 1. Then $\mathtt{T_1} \models \mathtt{T_2}$ or $\mathtt{T_1} \models \mathtt{T_3}$. Therefore, $\mathtt{T_1} \models \mathtt{T_2} \vee \mathtt{T_3}$ under Definition 1.*
- *Case $\mathtt{T_2} \vee \mathtt{T_3} \leq \mathtt{T_1}$: By S-UNION2, we have $\mathtt{T_2} \leq \mathtt{T_1} \rfloor \mathcal{C}$ and $\mathtt{T_3} \leq \mathtt{T_1} \rfloor \mathcal{C}$ and, hence, $\mathtt{T_2} \leq \mathtt{T_1} \rfloor \emptyset$ and $\mathtt{T_3} \leq \mathtt{T_1} \rfloor \emptyset$ by Lemma 1. Then $\mathtt{T_2} \models \mathtt{T_1}$ and $\mathtt{T_3} \models \mathtt{T_1}$. Therefore, $\mathtt{T_2} \vee \mathtt{T_3} \models \mathtt{T_1}$ under Definition 1.*
- *Case $\{\mathtt{T_1\ f_1} \ldots, \mathtt{T_i} \vee \mathtt{T_i'\ f_i}, \ldots, \mathtt{T_n\ f_n}\} \leq \mathtt{T}'$: By S-UNION3 and Lemma 1, $\{\mathtt{T_1\ f_1} \ldots, \mathtt{T_i\ f_i}, \ldots, \mathtt{T_n\ f_n}\} \leq \mathtt{T}' \rfloor \emptyset$ and $\{\mathtt{T_1\ f_1} \ldots, \mathtt{T_i'\ f_i}, \ldots, \mathtt{T_n\ f_n}\} \leq \mathtt{T}' \rfloor \emptyset$. Then, $\{\mathtt{T_1\ f_1} \ldots, \mathtt{T_i} \vee \mathtt{T_i'\ f_i}, \ldots, \mathtt{T_n\ f_n}\} \models \mathtt{T}'$ under Definition 1.*
- *Case $\overline{\mathtt{T}} \rightarrow \mathtt{T} \leq \overline{\mathtt{S}} \rightarrow \mathtt{S}$: By S-FUN, we have $\mathtt{T} \leq \mathtt{S} \rfloor \mathcal{C}$ and $\overline{\mathtt{S} \leq \mathtt{T} \rfloor \mathcal{C}}$ and, hence, $\mathtt{T} \leq \mathtt{S} \rfloor \emptyset$ and $\overline{\mathtt{S} \leq \mathtt{T} \rfloor \emptyset}$ by Lemma 1. It follows from Definition 1 that $\mathtt{T} \models \mathtt{S}$ and $\overline{\mathtt{S} \models \mathtt{T}}$ Assume $\overline{\mathtt{T}} \rightarrow \mathtt{T} \models \overline{\mathtt{V}} \rightarrow \overline{\mathtt{V}}$. By Definition 1, we have that $\overline{\mathtt{V} \models \mathtt{T}}$. Then, it follows that $\overline{\mathtt{V} \models \mathtt{S}}$. Hence, $\overline{\mathtt{S}} \rightarrow \mathtt{S} \models \overline{\mathtt{V}} \rightarrow \overline{\mathtt{V}}$.*

□

Theorem 2 (Subtype Completeness) *Let $\mathtt{T}$ and $\mathtt{T}'$ be types where $\mathtt{T} \models \mathtt{T}'$. Then, $\mathtt{T} \leq \mathtt{T}'$.*

Proof 3 *By induction on the structure of $\mathtt{T}$ and $\mathtt{T}'$. The induction hypothesis is that if $\mathtt{T_1} \models \mathtt{T_2}$ holds for some substructure of $\mathtt{T}$ and (resp.) $\mathtt{T}'$, then $\mathtt{T_1} \leq \mathtt{T_2} \rfloor \emptyset$. :*

- *Case $\mathtt{void} \models \mathtt{T}'$: Straightforward since $\mathtt{void} \leq \mathtt{T}'$ by S-VOID.*
- *Case $\mathtt{any} \models \mathtt{T}'$: By Definition 1, $\mathtt{any}$ accepts every value. Therefore, $\mathtt{T}'$ must accept every value and, hence, $\mathtt{T}' = \mathtt{any}$. Thus, $\mathtt{any} \leq \mathtt{any}$ by S-REFLEX.*
- *Case $\mathtt{null} \models \mathtt{T}'$: By Definition 1, $\mathtt{null}$ can only accept value $\mathtt{null}$. Therefore, either $\mathtt{T}' \equiv \mathtt{null}$ or $\mathtt{T}' \equiv \mathtt{null} \vee \mathtt{T_1}$ (for some $\mathtt{T_1}$). Then, $\mathtt{T} \leq \mathtt{T}'$ by S-NULL and/or S-UNION1.*
- *Case $[\mathtt{T_1}] \models [\mathtt{T_2}]$: By induction hypothesis, $\mathtt{T_1} \leq \mathtt{T_2}$ and, hence, $\mathtt{T} \leq \mathtt{T}'$ by S-LIST and S-UNION1.*
- *Case $\overline{\mathtt{T}} \rightarrow \mathtt{T} \models \overline{\mathtt{S}} \rightarrow \mathtt{S}$: By Definition 1, $\mathtt{T} \models \mathtt{S}$ and, by inductive hypothesis, $\mathtt{T} \leq \mathtt{S}$. It follows from Definition 1 that $\forall \overline{\mathtt{V}}.[\overline{\mathtt{V}} \models \mathtt{T_i} \implies \overline{\mathtt{V}} \models \mathtt{S_i}]$ (for $\mathtt{1} \leq \mathtt{i} \leq \mathtt{n}$) and, hence, $\overline{\mathtt{S} \models \mathtt{T}}$. Finally, $\overline{\mathtt{T} \geq \mathtt{S}}$ by inductive hypothesis and, hence, $\mathtt{T} \leq \mathtt{T}'$ by S-FUN.*
- *Case $\{\overline{\mathtt{T\ n}}\} \models \{\overline{\mathtt{S\ m}}\}$: By Definition 1, $\mathtt{m} \leq \mathtt{n}$. By induction hypothesis, $\overline{\mathtt{T} \leq \mathtt{S}}$ and, hence, $\mathtt{T} \leq \mathtt{T}'$ by S-REC.*
- *Case $\mathtt{T_1} \vee \mathtt{T_2} \models \mathtt{T}'$: By Definition 1, $\mathtt{T_1} \models \mathtt{T}'$ and $\mathtt{T_2} \models \mathtt{T}'$. By induction hypothesis, $\mathtt{T_1} \leq \mathtt{T}'$ and $\mathtt{T_2} \leq \mathtt{T}'$. Then, $\mathtt{T} \leq \mathtt{T}'$ by S-UNION2.*
- *Case $[\mathtt{T_1}] \models \mathtt{T_2} \vee \mathtt{T_3}$: If $\mathtt{T} \models \mathtt{T_2}$ or $\mathtt{T} \models \mathtt{T_3}$, then $\mathtt{T} \leq \mathtt{T_2} \vee \mathtt{T_3}$ by S-UNION1. Assume (for contradiction) that $\mathtt{T} \not\models \mathtt{T_2}$ and $\mathtt{T} \not\models \mathtt{T_3}$. It follows from Definition 1 that $\mathtt{T_2} = [\mathtt{T_4}]$, $\mathtt{T_3} = [\mathtt{T_5}]$ and, hence, $\mathtt{T_1} \models \mathtt{T_4} \vee \mathtt{T_5}$. This gives the contradiction as $[\mathtt{T_1}]$ will accept a list containing values from both $\mathtt{T_4}$ and $\mathtt{T_5}$ which neither $[\mathtt{T_4}]$ nor $[\mathtt{T_5}]$ could accept.*
- *Case $\overline{\mathtt{S}} \rightarrow \mathtt{S} \models \mathtt{T_1} \vee \mathtt{T_2}$: If $\mathtt{T} \models \mathtt{T_1}$ or $\mathtt{T} \models \mathtt{T_2}$, then $\mathtt{T} \leq \mathtt{T_1} \vee \mathtt{T_2}$ by S-UNION1. Assume (for contradiction) that $\mathtt{T} \not\models \mathtt{T_1}$ and $\mathtt{T} \not\models \mathtt{T_2}$. Let $\mathtt{T_1} = \overline{\mathtt{W}} \rightarrow \mathtt{W}$ and $\mathtt{T_2} = \overline{\mathtt{U}} \rightarrow \mathtt{U}$. Then, the contradiction arises in the same manner as for lists. For example, consider return values. It follows from Definition 1 that $\mathtt{S} \models \mathtt{W} \vee \mathtt{U}$. Then, $\mathtt{T}$ will accept a function with return values in both $\mathtt{W}$ and $\mathtt{U}$ which neither $\mathtt{T_1}$ nor $\mathtt{T_2}$ could accept.*
- *Case $\{\overline{\mathtt{S\ f}}\} \models \mathtt{T_1} \vee \mathtt{T_2}$: If $\mathtt{T} \models \mathtt{T_1}$ or $\mathtt{T} \models \mathtt{T_2}$, then $\mathtt{T} \leq \mathtt{T_1} \vee \mathtt{T_2}$ by S-UNION1. Assume $\mathtt{T} \not\models \mathtt{T_1}$ and $\mathtt{T} \not\models \mathtt{T_2}$. Then, some $\mathtt{S_i} = \mathtt{T_3} \vee \mathtt{T_4}$ must exist. Let $\mathtt{W} = \{\overline{\mathtt{W\ f}}\}$ where $\mathtt{W_i} = \mathtt{T_3}$ and $\forall_{\mathtt{j} \neq \mathtt{i}}.[\mathtt{W_j} = \mathtt{S_j}]$. Construct $\mathtt{W}'$ similarly, except with $\mathtt{W_i'} = \mathtt{T_4}$. It follows immediately that $\mathtt{W} \models \mathtt{T_1} \vee \mathtt{T_2}$ and $\mathtt{W}' \models \mathtt{T_1} \vee \mathtt{T_2}$. Hence, $\mathtt{T} \leq \mathtt{T}'$ by the inductive hypothesis and S-UNION3.*

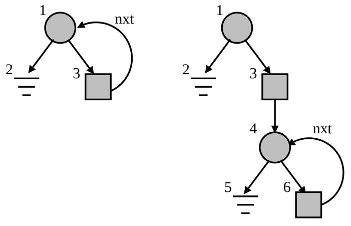
□

3.4 Representation

Another important question, from an implementation perspective, is how types can be represented efficiently. In particular, we desire a compact representation where equivalence testing is efficient[1]. Our starting point is to provide a *graph-based* definition of types:

Definition 7 (Type) *A* type *is a triple (V, E, K), where V is a set of vertices, E is a set of directed, labelled edges of the form $v \overset{\ell}{\to} w$, and K gives the kind of each vertex (i.e. NULL, INT, LIST, RECORD, UNION, etc).*

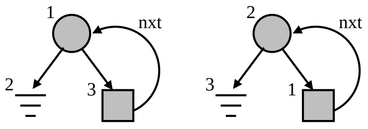
In this definition, the labels ℓ correspond either to field names or ϵ (i.e. no label). Thus, the recursive type $\mu X.(\texttt{null} \vee \{\texttt{int dt}, X\ \texttt{nxt}\})$ and its first unfolding can be viewed as follows:



The graph on the right represents a single unfolding of that on the left. Here, circular nodes represent unions, square nodes represent records, and **`null`** is indicated by ground (for brevity, we omit field `dt`).

Definition 8 (Minimal Form) *Let* $\texttt{T}\!\downarrow$ *denote a* minimal form *of a type* $\texttt{T}$ *such that, for any* $\texttt{T}' \equiv \texttt{T}$*, it follows that* $|\texttt{T}\!\downarrow| \leq |\texttt{T}'|$.

To compute $\texttt{T}\!\downarrow$ we can apply standard algorithms for minimising finite automata [49, 50, 51]. Observe that, for any given type, there may exist multiple minimal forms [52]. The following illustrates:



$$
\begin{aligned}
\texttt{T}_1 &= (\{1,2,3\}, \{1 \overset{\epsilon}{\to} 2, 1 \overset{\epsilon}{\to} 3, 3 \overset{nxt}{\to} 1\}, \{1 \mapsto \text{UNION}, 2 \mapsto \text{NULL}, 3 \mapsto \text{RECORD}\}) \\
\texttt{T}_2 &= (\{1,2,3\}, \{2 \overset{\epsilon}{\to} 3, 2 \overset{\epsilon}{\to} 1, 1 \overset{nxt}{\to} 2\}, \{1 \mapsto \text{RECORD}, 2 \mapsto \text{UNION}, 3 \mapsto \text{NULL}\})
\end{aligned}
$$

Whilst it is clear that $\texttt{T}_1 \equiv \texttt{T}_2$, it is also the case that $\texttt{T}_1 \neq \texttt{T}_2$. Thus, minimising a type does not, by itself, guarantee a canonical form.

Definition 9 (Canonical Form) *Let* $\hat{\texttt{T}}$ *denote the* canonical form *of a type* $\texttt{T}$ *such that, for any* $\texttt{T}_1$ *and* $\texttt{T}_2$ *where* $\texttt{T}_1 \equiv \texttt{T}_2$*, it follows that* $\hat{\texttt{T}}_1 = \hat{\texttt{T}}_2$.

This notion of a canonical representation is similar to that found in the study of graph isomorphism, where *canonical labellings* are used by the most efficient algorithms for determining isomorphism [53, 54, 55, 56]. To compute the canonical form of a type $\texttt{T}$ we first minimise it, and then determine a canonical labelling. Intuitively, the idea behind a canonical labelling is to ensure the labels given to vertices are always the same for equivalent types. To get a rough first approximation, we can sort vertices by type. For example, sorting according to NULL < RECORD < UNION gives the following canonical representation of $\texttt{T}_1$ and $\texttt{T}_2$:

$$
\begin{aligned}
\texttt{T}_3 \;=\; & (\{1,2,3\}, \{3 \overset{\epsilon}{\to} 1, 3 \overset{\epsilon}{\to} 2, 2 \overset{nxt}{\to} 3\}, \\
& \{1 \mapsto \text{NULL}, 2 \mapsto \text{RECORD}, 3 \mapsto \text{UNION}\})
\end{aligned}
$$

For simplicity, we will not discuss efficient algorithms for computing the canonical representation of a type. However, we will demonstrate that it exists and is always computable by a reduction to coloured graph isomorphism:

Definition 10 (Coloured Digraph) *A coloured digraph is a triple (V, E, C) where V is a set of vertices, E a set of directed edges of the form $v \to w$ and C maps every vertex to a colour taken from $\{1, \ldots, |V|\}$.*

[1]This stems from the fact that FW's typing algorithm employs fixed-point iteration and, in practice, this generates a lot of equivalence tests (see §4.4)

Lemma 2 (Canonical Form) *For any type* $\mathtt{T}$*, a canonical form* $\hat{\mathtt{T}}$ *can be computed such that, for any* $\mathtt{T}_1$ *and* $\mathtt{T}_2$ *where* $\mathtt{T}_1 \equiv \mathtt{T}_2$*, it follows that* $\hat{\mathtt{T}}_1 = \hat{\mathtt{T}}_2$*.*

Proof 4 *By reduction to coloured graph isomorphism. W.L.O.G. assume* $\mathtt{T}$ *is minimised. Construct a coloured graph* (V_C, E_C, C_C) *from a type* (V_T, E_T, K_T) *as follows:*

- *Let* $V_C = V_T \cup \{\ell_{vw} \mid v \xrightarrow{\ell} w \in E_T \wedge \ell \neq \epsilon\}$.
- *Let* $E_C = \{v \rightarrow w \mid v \xrightarrow{\epsilon} w \in E_T\} \cup \{v \rightarrow \ell_{vw}, \ell_{vw} \rightarrow w \mid v \xrightarrow{\ell} w \in E_T\}$.
- *Let* $C_C = \{v \mapsto c(k) \mid K_T(v) = k \wedge v \in V_T\} \cup \{\ell_{vw} \mapsto c(\ell) \mid v \xrightarrow{\ell} w \in E_T\}$ *where* $c(\cdot)$ *maps kinds and labels to unique colours.*

Then, a canonical labelling of (V_C, E_C, C_C) *can be computed with well-known algorithms that use backtracking search to find a minimum lexicographic ordering of edges (e.g. [53, 54]).* □

3.5 Other Operations

An important requirement is that our subtype relation forms a complete lattice. For any two types $\mathtt{T}_1$ and $\mathtt{T}_2$, the least upper bound and greatest lower bound must be well-defined:

Definition 11 *The least upper bound of two types* $\mathtt{T}_1$ *and* $\mathtt{T}_2$*, denoted* $\mathtt{T}_1 \sqcup \mathtt{T}_2$*, is the least type* $\mathtt{T}_3$ *where* $\mathtt{T}_1 \leq \mathtt{T}_3$ *and* $\mathtt{T}_2 \leq \mathtt{T}_3$*.*

Definition 12 *The greatest lower bound of two types* $\mathtt{T}_1$ *and* $\mathtt{T}_2$*, denoted* $\mathtt{T}_1 \sqcap \mathtt{T}_2$*, is the greatest type* $\mathtt{T}_3$ *where* $\mathtt{T}_3 \leq \mathtt{T}_1$ *and* $\mathtt{T}_3 \leq \mathtt{T}_2$*.*

The least upper bound operation is needed to determine variable types at control-flow joins. Computing $\mathtt{T}_1 \sqcup \mathtt{T}_2$ is fairly straightforward as, by construction, it is equivalent to $\mathtt{T}_1 \vee \mathtt{T}_2$. The purpose of the greatest lower bound operation may be less apparent. Consider:

```
define MyType as int | [int] | [[int]]

int f(MyType x):
   if x is [any]:
       return |x|
   else:
       return x
```

The most precise type for x on the true branch is $[\mathtt{int}] \vee [[\mathtt{int}]]$. To update a variable's type after a type test, we *intersect* its type before the test with that used in the test. Or, in other words, we compute their greatest lower bound:

$$(\mathtt{int} \vee [\mathtt{int}] \vee [[\mathtt{int}]]) \;\sqcap\; [\mathtt{any}] \Longrightarrow [\mathtt{int}] \vee [[\mathtt{int}]]$$

Computing $\mathtt{T}_1 \sqcap \mathtt{T}_2$ is only slightly more involved than for $\mathtt{T}_1 \sqcup \mathtt{T}_2$. Recalling the correspondence between types and tree automata from §3.1, then $\mathtt{T}_1 \sqcap \mathtt{T}_2$ corresponds to the *intersection* of the tree automata described by $\mathtt{T}_1$ and that described by $\mathtt{T}_2$. Again, standard algorithms for computing this are known [52].

Another question is how the type of x on the false branch is determined. We use a third operation on types called *greatest difference* (analogous to set difference):

Definition 13 *The greatest difference of* $\mathtt{T}_1$ *and* $\mathtt{T}_2$*, denoted* $\mathtt{T}_1 - \mathtt{T}_2$*, is the greatest type* $\mathtt{T}_3$ *where* $\mathtt{T}_3 \leq \mathtt{T}_1$ *and* $\mathtt{T}_2 \sqcap \mathtt{T}_3 = \mathtt{void}$*.*

Using the greatest difference operation, we obtain the following type for x on the false branch:

$$(\mathtt{int} \vee [\mathtt{int}] \vee [[\mathtt{int}]]) - [\mathtt{any}] \Longrightarrow \mathtt{int}$$

From this it follows that the last statement of function f() above is type safe since, at this point, x has type **int**. Unfortunately, Definition 13 can only be approximated in FW. For example:

```
int f(any x):
   if x is int:
       ...
   else:
       ...
```

On the false branch, we compute the type $\mathtt{any} - \mathtt{int}$ for x; however, this yields any, since there is no way to encode anything more precise in our language of types. In practice, this does not seem a major concern, as such types are not expressible at the source level.

4 Syntax, Semantics and Typing

Figure 3 gives the syntax for FW. As usual, the overbar (as in e.g. $\overline{\mathtt{S}}$) is taken to indicate a list with appropriate separator(s). For simplicity, indentation is not explicitly described in the syntax, but is instead assumed. The notion of a runtime value (given by v) is very similar, but not identical, to that from §3.1. In fact, the only difference lies in the way function values are represented. Finally, for simplicity, we treat direct and indirect invocation with an identical statement. This means direct invocation corresponds to indirect invocation on a constant function value.

4.1 Semantics

A big-step operational semantics for FW is given for expressions in Figure 4, and a small-step semantics for statements in Figure 5. Here, Δ is the *runtime environment*, whilst $\mathtt{v}, \mathtt{w}, \mathtt{u}$ denote *runtime values*. A runtime environment Δ provides access to local variables in the current stack frame, whilst $\mathtt{push}()$ and $\mathtt{pop}()$ let us (resp.) create and destroy stack frames. Likewise, `funDecl(f)` gives the declaration of a function `f` (we do not consider overloading). Finally, `halt(v)` is taken to mean that the machine has halted, producing value v.

Since function invocation is via a statement, rather than an expression, it follows that expressions have *finite executions* (i.e. they cannot loop indefinitely). This means we can safely employ a big-step semantics for expressions, which simplifies the system (we return to discuss this later). In contrast, we must employ a small-step semantics for statements, as these can loop infinitely (i.e. through **while** loops or recursion).

4.2 Type Environments

Typing functions requires separate environments for each program point. We denote the environment for some point ℓ by Γ^{ℓ}, which captures the types of all variables immediately before the statement at that point. For example:

```
int f(int x):
    y = x + 1¹
    return y²
```

The typing environments for this function are:

$$\begin{array}{lcl}\Gamma^1 & = & \{\mathtt{x} \mapsto \mathtt{int}\} \\ \Gamma^2 & = & \{\mathtt{x} \mapsto \mathtt{int}, \mathtt{y} \mapsto \mathtt{int}\}\end{array}$$

Since y is undefined before line 1, it is absent from Γ^1. To define how environments are combined at join points in the control-flow graph, we introduce a corresponding subtype relation:

Definition 14 *Let* $\Gamma^{\ell 1}$ *and* $\Gamma^{\ell 2}$ *be typing environments. Then,* $\Gamma^{\ell 1} \leq \Gamma^{\ell 2}$ *iff* $\forall \mathtt{v} \in \boldsymbol{dom}(\Gamma^{\ell 2}).\big[\Gamma^{\ell 1}[\mathtt{v}] \leq \Gamma^{\ell 2}[\mathtt{v}]\big]$.

For example, the following orderings are trivially valid:

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

On the other hand, the following do not hold:

$$\begin{array}{rcl}\{\mathtt{v} \mapsto [\mathtt{int}]\} & \not\leq & \{\mathtt{v} \mapsto \mathtt{int}\} \\ \{\mathtt{v} \mapsto \mathtt{int}\} & \not\leq & \{\mathtt{v} \mapsto \mathtt{void}\}\end{array}$$

The following illustrates how type environments are combined:

```
int|[int] f(int x):
   if y > 0¹:
      y = [x]²
      z = 2³
   else:
      y = []⁴
   return y⁵
```

The question is, what type does y have in Γ^5? We know that y has [int] or [void] type, and we desire the *most precise environment* capturing this. This corresponds to the *least upper bound* of the type environments involved:

$$\begin{array}{lcl}\Gamma^5 & = & \{\mathtt{x} \mapsto \mathtt{int}, \mathtt{y} \mapsto [\mathtt{int}], \mathtt{z} \mapsto \mathtt{int}\} \sqcup \{\mathtt{x} \mapsto \mathtt{int}, \mathtt{y} \mapsto [\mathtt{void}]\} \\ & \hookrightarrow & \{\mathtt{x} \mapsto \mathtt{int}, \mathtt{y} \mapsto [\mathtt{int}]\}\end{array}$$

Here, $\Gamma^5[\mathtt{y}] = [\mathtt{int}]$ as $[\mathtt{int}] \vee [\mathtt{void}] \equiv [\mathtt{int}]$. Furthermore, z cannot be included in Γ^5 as it is not defined on both branches.

Syntax:

$$
\begin{array}{lcl}
\mathtt{F} & ::= & \mathtt{T\ f(\overline{T\ n})\ :\ S} \\
\mathtt{S} & ::= & \mathtt{n = e} \mid \mathtt{n.f = e} \mid \mathtt{n[e_1] = e_2} \mid \mathtt{n = e(\overline{e})} \mid \mathtt{return\ e} \mid \mathtt{if\ n\ is\ T :\ S\ [else\ S]} \\
 & & \mid \mathtt{while\ e_1 < e_2 :\ S} \mid \mathtt{S_1; S_2} \\
\mathtt{e} & ::= & \mathtt{v} \mid \mathtt{n} \mid \mathtt{e.f} \mid \mathtt{e_1[e_2]} \mid \mathtt{[\overline{e}]} \mid \mathtt{\{\overline{n : e}\}} \\
\mathtt{v} & ::= & \mathtt{null} \mid \mathtt{i} \mid \mathtt{[\overline{v}]} \mid \mathtt{\{\ \overline{n : v}\ \}} \mid \mathtt{\&f}
\end{array}
$$

Figure 3: Syntax for Featherweight Whiley. Here, $\mathtt{n}$ represents variable identifiers, whilst $\mathtt{i} \in \mathcal{I}$ represent integer constants.

Expression Reductions:

$$
\frac{\overline{\mathtt{e} \leadsto \mathtt{v}}}{\mathtt{\{\overline{n : e}\}} \leadsto \mathtt{\{\overline{n : v}\}}}
\qquad
\frac{\overline{\mathtt{e} \leadsto \mathtt{v}}}{\mathtt{[\overline{e}]} \leadsto \mathtt{[\overline{v}]}}
\qquad
\begin{array}{r}\text{(R-RECORD,}\\ \text{R-LIST)}\end{array}
$$

$$
\frac{\begin{array}{c}\mathtt{e_2} \leadsto \mathtt{i}\\ 0 \le \mathtt{i} \le \mathtt{n}\\ \mathtt{e_1} \leadsto \mathtt{[v_0, \ldots, v_n]}\end{array}}{\mathtt{e_1[e_2]} \leadsto \mathtt{v_i}}
\qquad
\frac{\begin{array}{c}\mathtt{e_2} \leadsto \mathtt{i}\\ (\mathtt{i} < 0 \vee \mathtt{i} > \mathtt{n})\\ \mathtt{e_1} \leadsto \mathtt{[v_0, \ldots, v_n]}\end{array}}{\mathtt{e_1[e_2]} \leadsto \mathbf{err}}
\qquad
\begin{array}{r}\text{(R-LACCESS1,}\\ \text{R-LACCESS2)}\end{array}
$$

$$
\frac{\mathtt{e} \leadsto \mathtt{\{\overline{n : v}\}} \quad \mathtt{n_i} = \mathtt{f}}{\mathtt{e.f} \leadsto \mathtt{v_i}}
\qquad \text{(R-FACCESS)}
$$

Figure 4: Semantics for expressions in FW.

Statement Reductions:

$$
\frac{\Delta \vdash \mathtt{e} \leadsto \mathtt{v}}{\langle \Delta, \mathtt{n = e\ ;\ S} \rangle \longrightarrow \langle \Delta[\mathtt{n} \mapsto \mathtt{v}], \mathtt{S} \rangle}
\qquad \text{(R-VASSIGN)}
$$

$$
\frac{\begin{array}{c}\Delta \vdash \mathtt{e} \leadsto \mathtt{v}\\ \Delta(\mathtt{n}) = \mathtt{\{\overline{n : v}\}} \quad \mathtt{v'} = \mathtt{\{\overline{n : v}\}}[\mathtt{f} \mapsto \mathtt{v}]\end{array}}{\langle \Delta, \mathtt{n.f = e\ ;\ S} \rangle \longrightarrow \langle \Delta[\mathtt{n} \mapsto \mathtt{v'}], \mathtt{S} \rangle}
\qquad \text{(R-FASSIGN)}
$$

$$
\frac{\begin{array}{c}\Delta(\mathtt{n}) = \mathtt{[u_0, \ldots, u_n]} \quad \Delta \vdash \mathtt{e_2} \leadsto \mathtt{v_1}\\ \Delta \vdash \mathtt{e_1} \leadsto \mathtt{i : int} \quad 0 \le \mathtt{i} \le \mathtt{n}\\ \mathtt{v_2} = \mathtt{[u_0, \ldots, u_{i-1}, v_1, u_{i+1}, \ldots, u_n]}\end{array}}{\langle \Delta, \mathtt{n[e_1] = e_2\ ;\ S} \rangle \longrightarrow \langle \Delta[\mathtt{n} \mapsto \mathtt{v_2}], \mathtt{S} \rangle}
\qquad \text{(R-LASSIGN)}
$$

$$
\frac{\Delta_1(\mathtt{v}) \in \mathtt{T}}{\langle \Delta_1, \mathtt{if\ v\ is\ T\!:\ \{S_1\}\ else\ \{S_2\}\ ;\ S_3} \rangle \longrightarrow \langle \Delta_2, \mathtt{S_1\ ;\ S_3} \rangle}
\qquad \text{(R-IF1)}
$$

$$
\frac{\Delta_1(\mathtt{v}) \notin \mathtt{T}}{\langle \Delta_1, \mathtt{if\ v\ is\ T\!:\ \{S_1\}\ else\ \{S_2\}\ ;\ S_3} \rangle \longrightarrow \langle \Delta_2, \mathtt{S_2\ ;\ S_3} \rangle}
\qquad \text{(R-IF2)}
$$

$$
\frac{\begin{array}{c}\Delta \vdash \mathtt{e_1} \leadsto \mathtt{v_1 : int}\\ \Delta \vdash \mathtt{e_2} \leadsto \mathtt{v_2 : int} \quad \mathtt{v_1} \le \mathtt{v_2}\end{array}}{\begin{array}{l}\langle \Delta, \mathtt{while\ e_1 \le e_2 :\ \{S_1\}\ ;\ S_2} \rangle\\ \longrightarrow \langle \Delta, \mathtt{S_1; while\ e_1 \le e_2 :\ \{S_1\}\ ;\ S_2} \rangle\end{array}}
\qquad \text{(R-WHILE1)}
$$

$$
\frac{\begin{array}{c}\Delta \vdash \mathtt{e_1} \leadsto \mathtt{v_1 : int}\\ \Delta \vdash \mathtt{e_2} \leadsto \mathtt{v_2 : int} \quad \mathtt{v_1} \not\le \mathtt{v_2}\end{array}}{\langle \Delta, \mathtt{while\ e_1 \le e_2 :\ \{S_1\}\ ;\ S_2} \rangle \longrightarrow \langle \Delta, \mathtt{S_2} \rangle}
\qquad \text{(R-WHILE2)}
$$

$$
\frac{\begin{array}{c}\Delta \vdash \mathtt{e} \leadsto \mathtt{\&f} \quad \overline{\mathtt{e} \leadsto \mathtt{v}}\\ \mathtt{funDecl(f)} = \mathtt{T\ f(\overline{T\ n})\ :\ S_2}\\ \Delta_2 = \mathtt{push}(\Delta_1, \mathtt{S_1}, \mathtt{n})\end{array}}{\langle \Delta_1, \mathtt{n = e(\overline{e})\ ;\ S_1} \rangle \longrightarrow \langle \Delta_2[\overline{\mathtt{n} \mapsto \mathtt{v}}], \mathtt{S_2} \rangle}
\qquad \text{(R-INVOKE)}
$$

$$
\frac{\Delta_1 \vdash \mathtt{e} \leadsto \mathtt{v} \quad \Delta_2, \mathtt{S_2}, \mathtt{n} = \mathtt{pop}(\Delta_1)}{\langle \Delta_1, \mathtt{return\ e\ ;\ S_1} \rangle \longrightarrow \langle \Delta_2[\mathtt{n} \mapsto \mathtt{v}], \mathtt{S_2} \rangle}
\qquad \text{(R-RETURN)}
$$

$$
\frac{\Delta \vdash \mathtt{e} \leadsto \mathtt{v} \quad \epsilon = \mathtt{pop}(\Delta)}{\langle \Delta, \mathtt{return\ e\ ;\ S} \rangle \longrightarrow \mathtt{halt(v)}}
\qquad \text{(R-HALT)}
$$

Figure 5: Small-step operational semantics for statements in FW.

Expression Typing:

$$\frac{}{\Gamma \vdash \texttt{null} : \texttt{null}} \qquad \frac{\texttt{i} \in \mathbb{I}}{\Gamma \vdash \texttt{i} : \texttt{int}} \qquad \text{(T-NULL, T-INT)}$$

$$\frac{\texttt{funType}(\texttt{f}) = \overline{\texttt{T}} \to \texttt{T}}{\vdash \&\texttt{f} : \overline{\texttt{T}} \to \texttt{T}} \qquad \frac{\{\texttt{x} \mapsto \texttt{T}\} \in \Gamma}{\Gamma \vdash \texttt{x} : \texttt{T}} \qquad \text{(T-FUN, T-VAR)}$$

$$\frac{\Gamma \vdash \texttt{e}_0 : \texttt{T}_0, \ldots, \Gamma \vdash \texttt{e}_\texttt{n} : \texttt{T}_\texttt{n}}{\Gamma \vdash \{\overline{\texttt{n} : \texttt{e}}\} : \{\overline{\texttt{T n}}\}} \qquad \text{(T-RECORD)}$$

$$\frac{\Gamma \vdash \texttt{e}_0 : \texttt{T}_0, \ldots, \Gamma \vdash \texttt{e}_\texttt{n} : \texttt{T}_\texttt{n}}{\Gamma \vdash [\overline{\texttt{e}}] : [\texttt{T}_0 \vee \ldots \vee \texttt{T}_\texttt{n}]} \qquad \text{(T-LIST)}$$

$$\frac{\Gamma \vdash \texttt{e}_1 : [\texttt{T}] \quad \Gamma \vdash \texttt{e}_2 : \texttt{int}}{\Gamma \vdash \texttt{e}_1[\texttt{e}_2] : \texttt{T}} \qquad \text{(T-LACCESS)}$$

$$\frac{\Gamma \vdash \texttt{e} : \texttt{T} \quad \mathbf{ert}(\texttt{T}) = \{\ldots, \texttt{T}_\texttt{i}\ \texttt{f}, \ldots\}}{\Gamma \vdash \texttt{e.f} : \texttt{T}_\texttt{i}} \qquad \text{(T-FACCESS)}$$

Effective Record Type:

$$\frac{}{\mathbf{ert}(\{\overline{\texttt{T f}}\}) = \{\overline{\texttt{T f}}\}}$$

$$\frac{\begin{array}{c}\texttt{T}'_1 = \mathbf{ert}(\texttt{T}_1) = \{\overline{\texttt{T f}}\} \quad \texttt{T}'_2 = \mathbf{ert}(\texttt{T}_2) = \{\overline{\texttt{T}'\ \texttt{f}'}\} \\ \texttt{F} = \mathbf{dom}(\texttt{T}'_1) \cap \mathbf{dom}(\texttt{T}'_2) \\ \texttt{T}_3 = \{\texttt{T f} \mid \texttt{f} \in \texttt{F} \wedge \texttt{T} = \texttt{T}'_1[\texttt{f}] \vee \texttt{T}'_2[\texttt{f}]\}\end{array}}{\mathbf{ert}(\texttt{T}_1 \vee \texttt{T}_2) = \texttt{T}_3}$$

Figure 6: Typing rules for expressions in Featherweight Whiley.

4.3 Typing Expressions

The typing rules for expressions are given in Figure 6. These are presented using judgements of the form $\Gamma \vdash \texttt{e} : \texttt{T}$, which are taken to mean that under typing environment Γ, expression e has type T. Here, $\mathbf{funType}(\texttt{f})$ gives the static type of function f.

The *effective record type (ERT)* for type T is given by $\texttt{ert}(\texttt{T})$. This is necessary for handling unions of records which have fields in common. The following illustrates:

$$\begin{aligned}
&\texttt{ert}\Big(\{\texttt{int x}, [\texttt{int}]\ \texttt{dt}\} \vee \{\texttt{any x}\}\Big) = \{\texttt{any x}\} \\
&\texttt{ert}\Big(\{\texttt{int x}\} \vee \{[\texttt{int}]\ \texttt{x}, \texttt{int cd}\}\Big) = \{(\texttt{int} \vee [\texttt{int}])\ \texttt{x}\} \\
&\texttt{ert}\Big(\{\texttt{int x}, [\texttt{int}]\ \texttt{dt}\} \vee [\texttt{int}]\Big) = \{\} \\
&\texttt{ert}\Big(\mu\texttt{X}.\{\texttt{int x}, (\texttt{X} \vee \texttt{null})\ \texttt{n}\}\Big) \\
&= \texttt{ert}\Big(\{\texttt{int x}, (\mu\texttt{X}.\{\texttt{int x}, (\texttt{X} \vee \texttt{null})\ \texttt{n}\} \vee \texttt{null})\ \texttt{n}\}\Big) \\
&= \{\texttt{int x}, (\mu\texttt{X}.\{\texttt{int x}, ...\} \vee \texttt{null})\ \texttt{n}\}
\end{aligned}$$

For a union of records, the ERT is formed from those fields common to all by taking the LUB of their types. This identifies those fields (if any) which may be accessed on such a type.

Finally, rule T-LIST from Figure 6 indicates the type of a list constructor is the LUB of all element types:

$$\begin{array}{lcl}
\vdash \texttt{[1, [1]]} & : & [\texttt{int} \vee [\texttt{int}]] \\
\vdash \texttt{[[], [1]]} & : & [[\texttt{void}] \sqcup [\texttt{int}]] \Longrightarrow [[\texttt{int}]]
\end{array}$$

Using the LUB of the element types ensures we always get the most precise type for a list constructor.

4.4 Typing Statements

As expected, typing statements in FW takes a somewhat different approach than normal. When typing a statement, we need to describe its *effect* on the typing environment. The typing rules are given in Figure 7. These rules are presented as judgements of the form $\Gamma \vdash \texttt{S} : \Gamma'$. Here, Γ represents the typing environment immediately before S, whilst Γ' represents that which holds immediately after. Thus, the effect of statement S is captured in the difference between Γ and Γ'. To clarify this, consider the simplest example:

```
int f(string x, string y):
   x = 1$^1$
   return x$^2$
```

$\Gamma^1 = \{\mathtt{x}\mapsto\mathtt{string}, \mathtt{y}\mapsto\mathtt{string}\}$ gives the environment immediately before the assignment. Applying T-VASSIGN from Figure 7 yields the typing environment immediately after it, namely $\Gamma^2 = \{\mathtt{x}\mapsto\mathtt{int}, \mathtt{y}\mapsto\mathtt{string}\}$. Thus, the type of `x` is updated from `string` to `int` by the assignment.

Rule T-FASSIGN from Figure 7 makes use of the function $\mathbf{upert}(\mathtt{T}_1, \mathtt{f}, \mathtt{T}_2)$, which returns an updated version of $\mathtt{T}_1$ where field `f` now how has type $\mathtt{T}_2$. For unions of records this updates the field across all elements:

```
define LL as {int dt, LL nxt} | null

int f(LL link):
    if link != null$^1$:
        link.dt = [1]$^2$
        return 0$^3$
    ...
```

The typing environments determined for this function are:

$$
\begin{array}{lcl}
\Gamma^1 & = & \big\{\mathtt{link} \mapsto \mu\mathtt{X}.(\{\mathtt{int\ dt}, \mathtt{X\ nxt}\} \vee \mathtt{null})\big\} \\
\Gamma^2 & = & \big\{\mathtt{link} \mapsto \{\mathtt{int\ dt}, \mu\mathtt{X}.(\{\mathtt{int\ dt}, \mathtt{X\ nxt}\} \vee \mathtt{null})\ \mathtt{nxt}\}\big\} \\
\Gamma^3 & = & \big\{\mathtt{link} \mapsto \{[\mathtt{int}]\ \mathtt{dt}, \mu\mathtt{X}.(\{\mathtt{int\ dt}, \mathtt{X\ nxt}\} \vee \mathtt{null})\ \mathtt{nxt}\}\big\}
\end{array}
$$

Here, the **`null`** test removes the possibility of `link` being **`null`** in the true branch. Then, the assignment updates the type of field `dt` from **`int`** to `[`**`int`**`]`.
Rule T-LASSIGN also behaves somewhat unexpectedly:

```
[int|[int]] f([int] xs):
    if |xs| > 1:
        xs[0] = [1]
    return xs
```

Variable `xs` has type $[\mathtt{int} \vee [\mathtt{int}]]$ after the assignment as it now contains an element of $[\mathtt{int}]$ type. Of course, there are limitations on how well the type system can reason about list updates:

```
[[int]] f(int y):
    xs = [1]
    xs[0] = [1]
    return xs
```

Whilst `xs` must hold a value of type $[[\mathtt{int}]]$ at the return statement, the type system cannot prove this. The type of `xs` at this point is (conservatively) determined as $[\mathtt{int} \vee [\mathtt{int}]]$ and, hence, this program is rejected.

Rule T-RETURN uses $\mathbf{thisFunType}$ (which holds the enclosing function's type) to check the returned expression satisfies the required return type. The environment determined after a **`return`** statement is $\bot$ (the subtype of all typing environments). Consider the following:

```
int f(int x):
    if x == 0$^1$:
        return 0$^2$
    else:
        y = x + 1$^3$
    return y$^4$
```

The typing environment which holds before "**`return`** `y`" is:

$$\Gamma^4 = \bot \sqcup \{\mathtt{x} \mapsto \mathtt{int}, \mathtt{y} \mapsto \mathtt{int}\} \Longrightarrow \{\mathtt{x} \mapsto \mathtt{int}, \mathtt{y} \mapsto \mathtt{int}\}$$

The special environment $\bot$ is needed to ensure `y` retains its type at this point, since `y` is not defined on the true branch.

Finally, rule T-WHILE must determine the (least) fixed-point of the typing environment which holds within the body (as is common for flow-sensitive program analysis). This is more complicated than usual as, in some cases, fixed-point iteration is not guaranteed to terminate — we return to discuss this in the following section. This rule is also one reason we're interested in efficient equivalence testing of types (recall §3.4). This is because testing for the fixed-point generates a lot of type equivalence tests.

Statement Typing:

$$\frac{\Gamma_0 \vdash \mathtt{S}_1 : \Gamma_1 \quad \Gamma_1 \vdash \mathtt{S}_2 : \Gamma_2}{\Gamma_0 \vdash \mathtt{S}_1 ; \mathtt{S}_2 \ : \ \Gamma_2} \qquad \text{(T-SEQ)}$$

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{T}}{\Gamma \vdash \mathtt{n} = \mathtt{e} \ : \ \Gamma[\mathtt{n} \mapsto \mathtt{T}]} \qquad \text{(T-VASSIGN)}$$

$$\frac{\Gamma \vdash \mathtt{n} : \mathtt{T}_1 \quad \Gamma \vdash \mathtt{e} : \mathtt{T}_2 \quad \mathbf{ert}(\mathtt{T}_1) = \{\overline{\mathtt{T}\ \mathtt{f}}\}}{\Gamma \vdash \mathtt{n.f} = \mathtt{e} \ : \ \Gamma[\mathtt{n} \mapsto \mathbf{upert}(\mathtt{T}_1, \mathtt{f}, \mathtt{T}_2)]} \qquad \text{(T-FASSIGN)}$$

$$\frac{\Gamma \vdash \mathtt{n} : [\mathtt{T}_1] \quad \Gamma \vdash \mathtt{e}_1 : \mathtt{int} \quad \Gamma \vdash \mathtt{e}_2 : \mathtt{T}_2}{\Gamma \vdash \mathtt{n}[\mathtt{e}_1] = \mathtt{e}_2 \ : \ \Gamma[\mathtt{n} \mapsto [\mathtt{T}_1 \vee \mathtt{T}_2]]} \qquad \text{(T-LASSIGN)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{e} : \overline{\mathtt{T}} \to \mathtt{T} \\ \overline{\Gamma \vdash \mathtt{e} : \mathtt{T}'} \quad \overline{\mathtt{T}' \leq \mathtt{T}}\end{array}}{\Gamma \vdash \mathtt{n} = \mathtt{e}(\overline{\mathtt{e}}) \ : \ \Gamma[\mathtt{n} \mapsto \mathtt{T}]} \qquad \text{(T-INVOKE)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{e} : \mathtt{T} \quad \mathtt{T} \leq \mathtt{T}_r \\ \mathtt{thisFunType} = \overline{\mathtt{T}} \to \mathtt{T}_r\end{array}}{\Gamma \vdash \mathtt{return\ e} \ : \ \bot} \qquad \text{(T-RETURN)}$$

$$\frac{\begin{array}{c}\Gamma_0 \vdash \mathtt{v} : \mathtt{T}_1 \quad \Gamma_3 = \Gamma_1 \sqcup \Gamma_2 \\ \Gamma_0[\mathtt{v} \mapsto \mathtt{T}_1 \sqcap \mathtt{T}_2] \vdash \mathtt{S} : \Gamma_1 \quad \Gamma_0[\mathtt{v} \mapsto \mathtt{T}_1 - \mathtt{T}_2] \vdash \mathtt{S}' : \Gamma_2\end{array}}{\Gamma_0 \vdash \mathtt{if\ v\ is\ T}_2 \mathtt{:}\ \{\mathtt{S}\}\ \mathtt{else}\ \{\mathtt{S}'\} : \Gamma_3} \qquad \text{(T-IF)}$$

$$\frac{\begin{array}{c}\Gamma_0 \sqcup \Gamma_1 \vdash \mathtt{S} : \Gamma_1 \\ \Gamma_0 \sqcup \Gamma_1 \vdash \mathtt{e}_1 : \mathtt{int} \quad \Gamma_0 \sqcup \Gamma_1 \vdash \mathtt{e}_2 : \mathtt{int}\end{array}}{\Gamma_0 \vdash \mathtt{while\ e}_1 \leq \mathtt{e}_2\ \{\mathtt{S}\} : \Gamma_0 \sqcup \Gamma_1} \qquad \text{(T-WHILE)}$$

Update Effective Record Type:

$$\frac{\mathtt{T}_1 = \{\overline{\mathtt{T}\ \mathtt{f}}\}}{\mathbf{upert}(\mathtt{T}_1, \mathtt{f}_j, \mathtt{T}_2) = \mathtt{T}_1[\mathtt{f}_j \mapsto \mathtt{T}_2]}$$

$$\frac{\mathtt{T}_4 = \mathbf{upert}(\mathtt{T}_1, \mathtt{f}, \mathtt{T}_3) \vee \mathbf{upert}(\mathtt{T}_2, \mathtt{f}, \mathtt{T}_3)}{\mathbf{upert}(\mathtt{T}_1 \vee \mathtt{T}_2, \mathtt{f}, \mathtt{T}_3) = \mathtt{T}_4}$$

Figure 7: Typing rules for statements in FW.

5 Termination and Soundness

In this section, we prove two important properties for FW, namely: *soundness* and *termination*. Proving termination of FW's typing algorithm depends on showing the type environments $\Gamma_0, \ldots, \Gamma_n$ for rule T-WHILE are monotonically increasing and, hence, always reach a fixed point. Unfortunately, by itself, this is not enough as the lattice has infinite height — however, we show how widening can be used to ensure termination without any loss of precision. Soundness requires that every well-typed program, when given appropriate input, will execute without getting stuck.

5.1 Termination

Showing termination of the typing algorithm relies on showing that we have a join-semi lattice (i.e. of the subtype relation) and that the transfer functions (i.e. the statement typing rules) are monotonic. However, these properties alone are insufficient to show termination of the typing rule T-WHILE from Figure 7. This is because the lattice of types has infinite height and, hence, the fixed-point algorithm may ascend it forever. To address this, we identify the special cases where it may happen, and demonstrate that widening can be employed to ensure termination *without loss of precision*. That is, assuming widening is performed at the correct point, the resulting typing for any program *will be as precise as is possible*.

As an example, consider the following simple FW function:

```
define LoopyList as [int | LoopyList]

LoopyList loopy(int n):
   x = [0]^1
   while |x| < n:
       x[0] = x^2
   return x
```

This method will cause a naive fix-point implementation of rule T-WHILE from Figure 7 to iterate forever. The following illustrates the environments that would be produced:

$$\begin{array}{lcl}
\Gamma^1 & = & \{\mathtt{n} \mapsto \mathtt{int}, \mathtt{x} \mapsto [\mathtt{int}]\} \\
\Gamma^2 & = & \{\mathtt{n} \mapsto \mathtt{int}, \mathtt{x} \mapsto [\mathtt{int}]\} \\
\Gamma^2 & = & \{\mathtt{n} \mapsto \mathtt{int}, \mathtt{x} \mapsto [\mathtt{int} \vee [\mathtt{int}]]\} \\
\Gamma^2 & = & \{\mathtt{n} \mapsto \mathtt{int}, \mathtt{x} \mapsto [\mathtt{int} \vee [\mathtt{int} \vee [\mathtt{int}]]]\} \\
\Gamma^2 & = & \{\mathtt{n} \mapsto \mathtt{int}, \mathtt{x} \mapsto [\mathtt{int} \vee [\mathtt{int} \vee [\mathtt{int} \vee [\mathtt{int}]]]]\} \\
\ldots & &
\end{array}$$

We can see that the type of x is essentially growing on every iteration of T-WHILE. Informally, we refer to this behaviour as being *divergent* (with respect to typing). Now, the crucial observation is that we can, in fact, give a precise type for x in Γ^2, namely $\mu X.[\mathtt{int} \vee \mathtt{X}]$. Thus, the algorithm must infer this type from the divergent behaviour, as follows:

$$\begin{array}{lcl}
\Gamma^2 & = & \{\mathtt{n} \mapsto \mathtt{int}, \mathtt{x} \mapsto [\mathtt{int} \vee [\mathtt{int} \vee [\mathtt{int}]]]\} \\
\Gamma^2 & = & \{\mathtt{n} \mapsto \mathtt{int}, \mathtt{x} \mapsto [\mathtt{int} \vee [\mathtt{int} \vee [\mathtt{int} \vee [\mathtt{int}]]]]\} \\
\ldots & & \\
\Gamma^2 & = & \{\mathtt{n} \mapsto \mathtt{int}, \mathtt{x} \mapsto \mu X.[\mathtt{int} \vee \mathtt{X}]\} \\
\Gamma^2 & = & \{\mathtt{n} \mapsto \mathtt{int}, \mathtt{x} \mapsto [\mathtt{int} \vee \mu X.[\mathtt{int} \vee \mathtt{X}]]\}
\end{array}$$

At this point, the computation terminates because $\mu X.[\mathtt{int} \vee \mathtt{X}] \equiv [\mathtt{int} \vee \mu X.[\mathtt{int} \vee \mathtt{X}]]$ under Definition 3. The technique used here is often referred to as *widening* in the program analysis literature (see e.g. [57, 58, 59]) and, generally speaking, results in lost precision. However, in our setting, *widening does not lose precision* (although it is necessary to ensure termination).

Lemma 3 (Monotonicity) *Let* $\Gamma_1 \vdash \mathtt{S} : \Gamma_2$ *and* $\Gamma'_1 \vdash \mathtt{S} : \Gamma'_2$ *be typing environments. If* $\Gamma_1 \leq \Gamma'_1$*, then* $\Gamma_2 \leq \Gamma'_2$.

Proof 5 *Straightforward by inspection of Figure 7.* □

Definition 15 (Divergence) *Let* $\Gamma_1 \Longmapsto \mathtt{S}$ *denote a diverging typing problem where* $\Gamma_1 \vdash \mathtt{S} : \Gamma_2, \Gamma_2 \vdash \mathtt{S} : \Gamma_3, \ldots$ *is an infinite sequence such that* $\forall \mathtt{i}.[\Gamma_\mathtt{i} < \Gamma_{\mathtt{i}+1}]$.

Lemma 4 (Divergence) *For a diverging typing problem* $\Gamma_1 \Longmapsto \mathtt{S}$*, a variable* x *exists where* $\Gamma_\mathtt{i}(\mathtt{x}) < \Gamma_{\mathtt{i}+1}(\mathtt{x})$ *occurs infinitely often.*

Proof 6 *Straightforward as, otherwise, computation reaches a fixed point in finitely many steps as we have finitely many variables.* □

Definition 16 *Let* $\mathtt{T}_1[\![\mathtt{T}_2]\!]$ *indicate type* $\mathtt{T}_1$ *contains* $\mathtt{T}_2$ *as one (or more) subcomponents.*

Definition 17 (Induction Variable) *A diverging typing problem* $\Gamma_1 \Longmapsto \mathtt{S}$ *is said to diverge on a variable* x *if* $\Gamma_\mathtt{i}(\mathtt{x}) \equiv \mathtt{T}[\![\Gamma_\mathtt{j}(\mathtt{x})]\!]$ *for* $\mathtt{i} > \mathtt{j}$ *occurs infinitely often.*

Lemma 5 (Induction Variable) *Every diverging typing problem* $\Gamma_1 \Longmapsto \mathtt{S}$ *is diverging upon at least one variable* x.

Proof 7 *Let* $\mathtt{x_i}$ *denote those variables where* $\Gamma_\mathtt{i}(\mathtt{x_i}) < \Gamma_{\mathtt{i}+1}(\mathtt{x_i})$ *occurs infinitely often (Lemma 4). Then, there are infinitely many types generated for each* $\mathtt{x_i}$*. Furthermore, new types can only be generated from constants, control-flow merges, type tests and through statements of the form* $\mathtt{x_i}[\mathtt{e}_2] = \mathtt{e}_1$ *and* $\mathtt{x_i}.\mathtt{f} = \mathtt{e}_1$*. Since the program structure is fixed, the divergence must stem from one or more statements of the form* $\mathtt{x_i}[\mathtt{e}_1] = \mathtt{e}_2$ *and* $\mathtt{x_i}.\mathtt{f} = \mathtt{e}$*. An infinite number of types can only be generated from such statements through divergence on* $\mathtt{x_i}$. □

An interesting question is whether or not we can strengthen our notion of an induction variable to say $\Gamma_i(x) \equiv T[\![\Gamma_{i+1}(x)]\!]$ occurs infinitely often. We have found neither counter-example nor proof, but conjecture this is the case. However, one certainly cannot say that $\Gamma_i(x) \equiv T[\![\Gamma_{i+1}(x)]\!]$ for all $i \geq 1$, as the following illustrates:

```
x = [0]
y = 1
while ... :
    x[0] = y
    y = x
```

Here, $\Gamma_1(x) = \Gamma_2(x) = [\texttt{int}]$. Furthermore, Lemma 5 might appear to fall down on this example:

```
x = [[0]]
while ... :
    x[0][0] = x[0]
```

This is because the key property that $\Gamma_{i+1}(x) = T[\![\Gamma_i(x)]\!]$ does not appear to hold. However, the above example is not valid FW code! Rather, it's translation would be:

```
x = [[0]]
while ... :
    tmp = x[0]
    tmp[0] = x[0]
    x[0] = tmp
```

Thus, in this case, it follows that $\Gamma_2(\texttt{tmp}) = [\texttt{int} \vee \Gamma_1(\texttt{tmp})]$, $\Gamma_3(\texttt{tmp}) = [\texttt{int} \vee \Gamma_1(\texttt{tmp}) \vee \Gamma_2(\texttt{tmp})]$, etc.

Definition 18 (Least Widening) *Let $\Gamma_1 \Longmapsto S$ be a typing problem which diverges on x. Then, a* widening *of x is the least type T such that either $\Gamma_1[x \mapsto T] \vdash S : \Gamma_2$ or $\Gamma_1[x \mapsto T] \Longmapsto S$ no longer diverges on x.*

Lemma 6 (Least Widening) *For a typing problem $\Gamma_1 \Longmapsto S$ which diverges on x, there exists a least widening of x.*

Proof 8 *Following Lemma 5 we have $\Gamma_i(x) \equiv T[\![\Gamma_j(x)]\!]$ for $i > j$ infinitely often. Pick the smallest such i and j. Then, by construction, the least widening of x is $\mu X.\Gamma_i(x)[\![\Gamma_j(x) \mapsto X]\!]$.* □

Definition 18 indicates that a diverging typing problem may diverge on multiple independent variables. Widening just one of them does not necessarily eliminate all divergent behaviour. However, we can simply widen each individually until none remains.

We now return to consider the issue of showing that the least fixed point of the typing equations defined in Figure 7 is computable. The essential idea is that, for every non-divergent problem, we can just compute the fixed-point iteratively; but, for a divergent problem we immediately widen the diverging variable(s). The termination theorem states that a least solution to a (possibly diverging) typing problem $\Gamma_1 \vdash S : \Gamma_2$ can be computed:

Theorem 3 (Termination) *Given a typing environment Γ_1, and a statement S, the least solution Γ_2 which satisfies $\Gamma_1 \vdash S : \Gamma_2$ is computable.*

Proof 9 *Straightforward given Lemma 3, Lemma 4 and Lemma 6.* □

5.2 Soundness

In the following Lemmas, we use the term "closed expression" to represent an expression without free variables. Recall from §4.1 that our semantics for expressions given in Figure 4 guarantees finite execution traces. That is, expressions cannot execute indefinitely and will either produce a value, or halt with an error (indicated by **err**):

Lemma 7 (Finite Traces) *Let e_1 be an arbitrary closed expression. Then, either $e_1 \leadsto v$, or $e_1 \leadsto \mathbf{err}$.*

Proof 10 *Straightforward by inspection of Figure 4.* □

Essentially, Lemma 7 justifies our use of a big-step operational semantics for expressions, as it means we do not need to consider the case of non-terminating expressions (which is known to be difficult with a big-step operational semantics, and one important reason for using small-step [60]). As a result, our type-safety proof for expressions does not employ the usual notions of *progress* and *preservation*. Instead, we can prove type-safety directly as follows:

Lemma 8 (Effective Record Type) *Let* $\texttt{ert}(\texttt{T}) = \{\overline{\texttt{T f}}\}$ *for some type* $\texttt{T}$. *Then* $\texttt{T} \leq \{\overline{\texttt{T f}}\}$.

Proof 11 *Looking at Figure 6 there are two rules defining* $\texttt{ert}(\texttt{T})$. *The case for* $\texttt{ert}(\{\overline{\texttt{T n}}\})$ *follows immediately, whilst* $\texttt{ert}(\texttt{T}_1 \vee \texttt{T}_2)$ *follows if we observe that* $\texttt{T}_1 \leq \texttt{T}'_1$ *and* $\texttt{T}_2 \leq \texttt{T}'_2$ *(by induction), and* $\{\texttt{T}'_1, \texttt{T}'_2\} \leq \texttt{T}_3$. *This latter because* $\texttt{T}_3$ *includes only fields present in both, and determines the type of each field* $\texttt{f}$ *from the least upper bound of its type in* $\texttt{T}'_1$ *and* $\texttt{T}'_2$. □

Lemma 9 (Expression Safety) *Let* $\texttt{e}$ *be a well-typed, closed expression (i.e.* $\vdash \texttt{e} : \texttt{T}_\texttt{e}$*). Then,* $\texttt{e} \leadsto \texttt{v}$ *where* $\vdash \texttt{v} : \texttt{T}_\texttt{v}$ *and* $\texttt{T}_\texttt{v} \leq \texttt{T}_\texttt{e}$.

Proof 12 *By induction on the structure of* $\texttt{e} : \texttt{T}_\texttt{e}$. *The T-NULL, T-INT and T-REAL cases are immediate, since* $\texttt{e}$ *in these cases is a value.*

- *Case "*$\{\overline{\texttt{f} : \texttt{e}_\texttt{f}}\}$*": Induction hypothesis guarantees, for each field* $\texttt{f}$ *with initialiser* $\texttt{e}_\texttt{f} : \texttt{T}_\texttt{f}$, *that* $\overline{\texttt{e}_\texttt{f} \leadsto \texttt{v}_\texttt{f}}$ *where* $\vdash \texttt{v}_\texttt{f} : \texttt{T}'_\texttt{f}$ *and* $\texttt{T}'_\texttt{f} \leq \texttt{T}_\texttt{f}$. *Thus,* $\vdash \{\overline{\texttt{f} : \texttt{e}_\texttt{f}}\} \leadsto \{\overline{\texttt{f} : \texttt{v}_\texttt{f}}\} : \texttt{T}_\texttt{v}$ *and so* $\texttt{T}_\texttt{v} \leq \texttt{T}_\texttt{e}$ *since we have* $\{\overline{\texttt{T}'_\texttt{f}\ \texttt{f}}\} \leq \{\overline{\texttt{T}_\texttt{f}\ \texttt{f}}\}$ *by S-REC.*
- *Case "*$[\overline{\texttt{e}}]$*": Induction hypothesis guarantees, for each element expression* $\texttt{e}_\texttt{i} : \texttt{T}_\texttt{i}$, *that* $\texttt{e}_\texttt{i} \leadsto \texttt{v}_\texttt{i}$ *where* $\vdash \texttt{v}_\texttt{i} : \texttt{T}'_\texttt{i}$ *and* $\texttt{T}'_\texttt{i} \leq \texttt{T}_\texttt{i}$. *Thus,* $\texttt{T}_\texttt{v} \leq \texttt{T}_\texttt{e}$ *because* $\texttt{T}'_1\sqcup, \ldots, \sqcup\texttt{T}'_\texttt{n} \leq \texttt{T}_1\sqcup, \ldots, \sqcup\texttt{T}_\texttt{n}$ *by Definition 11.*
- *Case "*$\texttt{e}_1[\texttt{e}_2]$*": Induction hypothesis guarantees, for source expression* $\texttt{e}_1 : [\texttt{T}_\texttt{e}]$, *that* $\texttt{e}_1 \leadsto [\overline{\texttt{v}}]$ *where* $\vdash [\overline{\texttt{v}}] : [\texttt{T}']$ *and* $[\texttt{T}_\texttt{v}] \leq [\texttt{T}_\texttt{e}]$. *Likewise, for the index expression, it guarantees* $\texttt{e}_2 \leadsto \texttt{i}$ *where* $\vdash \texttt{i} : \texttt{int}$. *Thus, assuming the list access is not out-of-bounds, we have* $\texttt{T}_\texttt{v} \leq \texttt{T}_\texttt{e}$.
- *Case "*$\texttt{e}_1.\texttt{f}$*": Induction hypothesis guarantees, for source expression* $\texttt{e}_1 : \texttt{T}_1$, *that* $\texttt{e}_1 \leadsto \texttt{v}_1$ *where* $\vdash \texttt{v}_1 : \{\overline{\texttt{T}'_\texttt{f}\ \texttt{f}}\}$ *and* $\{\overline{\texttt{T}'_\texttt{f}\ \texttt{f}}\} \leq \texttt{T}_1$. *T-FACCESS also ensures that* $\texttt{ert}(\texttt{T}_1) = \{\overline{\texttt{T}_\texttt{f}\ \texttt{f}}\}$ *and, hence, it follows by Lemma 8 that* $\{\overline{\texttt{T}'_\texttt{f}\ \texttt{f}}\} \leq \texttt{T}_1 \leq \{\overline{\texttt{T}_\texttt{f}\ \texttt{f}}\}$. *Finally, by S-REC, we have* $\{\overline{\texttt{T}'_\texttt{f}\ \texttt{f}}\}.\texttt{f} \leq \{\overline{\texttt{T}_\texttt{f}\ \texttt{f}}\}.\texttt{f}$ *and, hence,* $\texttt{T}_\texttt{v} \leq \texttt{T}_\texttt{e}$.

□

At this point, we can now extend our notion of expression evaluation to include variables from the runtime environment, as follows:

Definition 19 (Expression Evaluation) *Let* Δ *be a runtime environment, and* $\texttt{e}$ *an arbitrary expression with free variables given by* $\texttt{freeVars}(\texttt{e}) = \overline{\texttt{x}}$. *If* $\texttt{e}[\overline{\texttt{x} \mapsto \Delta(\texttt{x})}] \leadsto \texttt{v}$ *where* $\texttt{v} : \texttt{T}$, *then we say* $\texttt{e}$ *evaluates to a value* v *of type* $\texttt{T}$ *under* Δ, *denoted by* $\Delta \vdash \texttt{e} \leadsto \texttt{v} : \texttt{T}$.

The following notion of a *safe abstraction* captures the relationship between type environments and their corresponding runtime environments:

Definition 20 (Safe Abstraction) *Let* Γ *be a typing environment and* Δ *a runtime environment. Then,* Γ *is a safe abstraction of* Δ, *denoted* $\Gamma \approx \Delta$, *iff* $\textbf{dom}(\Gamma) \subseteq \textbf{dom}(\Delta)$ *and, for all* $\texttt{x} \in \textbf{dom}(\Gamma)$ *where* $\Gamma(\texttt{x}) = \texttt{T}_\texttt{x}$, $\Delta(\texttt{x}) = \texttt{v}$ *and* $\vdash \texttt{v} : \texttt{T}_\texttt{v}$, *it holds that* $\texttt{T}_\texttt{v} \leq \texttt{T}_\texttt{x}$.

Observe that we cannot require $\textbf{dom}(\Gamma) = \textbf{dom}(\Delta)$, as might be expected, since runtime environments are the product of actual execution paths. Consider an **if** statement with a variable $\texttt{x}$ defined only on one branch. After the statement, $\texttt{x} \notin \Gamma$ since $\texttt{x}$ was not defined on both branches. However, if execution had proceeded through the branch where $\texttt{x}$ was defined, then we would have $\texttt{x} \in \Delta$.

Lemma 10 (Safe Join) *Let* $\Gamma_1 \approx \Delta_1$. *Then,* $\Gamma_1 \sqcup \Gamma_2 \approx \Delta_1$ *for any* Γ_2.

Proof 13 *Straightforward, since by construction* $\Gamma_1 \leq \Gamma_1 \sqcup \Gamma_2$ *and, hence,* $\forall\texttt{x}.[\Gamma_1(\texttt{x}) \leq (\Gamma_1 \sqcup \Gamma_2)(\texttt{x})]$ *holds by Definition 14.* □

Theorem 4 (Progress) *Let* Δ_1 *be a runtime environment and* Γ_1 *a typing environment where* $\Gamma_1 \approx \Delta_1$. *If* $\Gamma_1 \vdash \texttt{S} : \Gamma_2$, *then either* $\langle \Delta_1, \texttt{S}\ ;\ \texttt{S}' \rangle \longrightarrow \langle \Delta_2, \texttt{S}'' \rangle$ *or* $\langle \Delta_1, \texttt{S}\ ;\ \texttt{S}' \rangle \longrightarrow \texttt{halt}(\texttt{v})$.

Proof 14 *By induction on the structure of* $\texttt{S}$.

- *Case "*$\texttt{x} = \texttt{e}\ ;\ \texttt{S}'$*": Since* $\Gamma_1 \vdash \texttt{S} : \Gamma_2$ *and* $\Gamma_1 \approx \Delta_1$, *it follows by T-VASSIGN that* $\Gamma_1 \vdash \texttt{e} : \texttt{T}_\texttt{e}$ *and, by Lemma 9, that* $\Delta_1 \vdash \texttt{e} \leadsto \texttt{v}$. *Hence,* $\langle \Delta_1, \texttt{x} = \texttt{e}\ ;\ \texttt{S}' \rangle \longrightarrow \langle \Delta_1[\texttt{x} \mapsto \texttt{v}], \texttt{S}' \rangle$.

- *Case "*$\mathtt{x.f = e\ ;\ S'}$*": Since* $\Gamma_1 \vdash \mathtt{S} : \Gamma_2$ *and* $\Gamma_1 \approx \Delta_1$*, it follows by T-FASSIGN that* $\Gamma_1 \vdash \mathtt{x} : \mathtt{T_x}$ *where* $\mathbf{ert}(\mathtt{T_x}) = \{\overline{\mathtt{T\ f}}\}$ *and, hence,* $\Delta_1 \vdash \mathtt{x} \rightsquigarrow \{\overline{\mathtt{f : v}}\}$ *by Lemma 9. Likewise,* $\Gamma_1 \vdash \mathtt{e} : \mathtt{T_e}$ *and, by Lemma 9,* $\Delta_1 \vdash \mathtt{e} \rightsquigarrow \mathtt{v_1}$*. Hence,* $\langle \Delta_1, \mathtt{x.f = e\ ;\ S'} \rangle \longrightarrow \langle \Delta_1[\mathtt{x} \mapsto \mathtt{v_2}], \mathtt{S'} \rangle$*, where* $\mathtt{v_2} = \{\overline{\mathtt{f : v}}\}[\mathtt{f} \mapsto \mathtt{v_1}]$.

- *Case "*$\mathtt{x[e_1] = e_2\ ;\ S'}$*": Since* $\Gamma_1 \vdash \mathtt{S} : \Gamma_2$ *it follows by T-LASSIGN that* $\Gamma_1 \vdash \mathtt{e_1} : \mathtt{int}$*,* $\Gamma_1 \vdash \mathtt{e_2} : \mathtt{T_1}$ *and* $\Gamma_1 \vdash \mathtt{x} : [\mathtt{T_3}]$*. Thus,* $\Gamma_1 \approx \Delta_1$ *and Lemma 9 imply that* $\Delta_1 \vdash \mathtt{e_1} \rightsquigarrow \mathtt{v_1} : \mathtt{int}$*,* $\Delta_1 \vdash \mathtt{e_2} \rightsquigarrow \mathtt{v_2} : \mathtt{T_2}$ *and* $\Delta_1 \vdash \mathtt{x} \rightsquigarrow \mathtt{v_x}$ *(where* $\mathtt{v_x} = [\overline{\mathtt{v}}]$*). Thus,* $\langle \Delta_1, \mathtt{x[e_1] = e_2\ ;\ S'} \rangle \longrightarrow \langle \Delta_1[\mathtt{x} \mapsto \mathtt{v'_x}], \mathtt{S'} \rangle$*, where* $\mathtt{v'_x} = \mathtt{v_x}[\mathtt{v_1} \mapsto \mathtt{v_2}]$.

- *Case "*$\mathtt{return\ e\ ;\ S'}$*": Since* $\Gamma_1 \vdash \mathtt{S} : \Gamma_2$ *and* $\Gamma_1 \approx \Delta_1$*, it follows by T-RETURN that* $\Gamma_1 \vdash \mathtt{e} : \mathtt{T_e}$ *and, by Lemma 9, that* $\Delta_1 \vdash \mathtt{e} \rightsquigarrow \mathtt{v_1}$*. Hence,* $\langle \Delta_1, \mathtt{return\ e\ ;\ S'} \rangle \longrightarrow \langle \Delta_2, \mathtt{S''} \rangle$ *or* $\langle \Delta_1, \mathtt{return\ e} \rangle \longrightarrow \mathtt{halt(v)}$.

- *Case "*$\mathtt{if\ v \sim= T\!:\ S_1\ else\!:\ S_2}$*": Since* $\Gamma_1 \vdash \mathtt{S} : \Gamma_2$ *and* $\Gamma_1 \approx \Delta_1$*, it follows that* $\Delta_1 \vdash \mathtt{x} \rightsquigarrow \mathtt{v_x}$*. Thus, either T-IF1 or T-IF2 apply, leading to a transition of the form* $\langle \Delta_1, \mathtt{S} \rangle \longrightarrow \langle \Delta_1, \mathtt{S'} \rangle$.

- *Case "*$\mathtt{while\ e_1 \leq e_2\!:\ S}$*": Since* $\Gamma_1 \vdash \mathtt{S} : \Gamma_2$ *it follows by T-WHILE that* $\Gamma_1 \vdash \mathtt{e_1} : \mathtt{int}$ *and* $\Gamma_2 \vdash \mathtt{e_2} : \mathtt{int}$*. Then,* $\Gamma_1 \approx \Delta_1$ *implies that* $\Delta_1 \vdash \mathtt{e_1} \rightsquigarrow \mathtt{v_1} : \mathtt{int}$ *and* $\Delta_1 \vdash \mathtt{e_2} \rightsquigarrow \mathtt{v_2} : \mathtt{int}$ *by Lemma 9. Thus, either R-WHILE1 or R-WHILE2 apply, leading to a transition of the form* $\langle \Delta_1, \mathtt{S} \rangle \longrightarrow \langle \Delta_2, \mathtt{S'} \rangle$.

- *Case "*$\mathtt{n = e(\overline{e})\ ;\ S'}$*": Since* $\Gamma_1 \vdash \mathtt{S} : \Gamma_2$ *and* $\Gamma_1 \approx \Delta_1$*, it follows by T-INVOKE that* $\Delta_1 \vdash \mathtt{e} : \overline{\mathtt{T}} \rightarrow \mathtt{T}$*, and that* $\overline{\Delta_1 \vdash \mathtt{e} : \mathtt{S}}$ *where* $\overline{\mathtt{S} \leq \mathtt{T}}$*. Hence, by R-INVOKE we have* $\langle \Delta_1, \mathtt{n = e(\overline{e})\ ;\ S'} \rangle \longrightarrow \langle \Delta_2, \mathtt{S''} \rangle$.

$\square$

Theorem 5 (Preservation) *Let* $\langle \Delta_1, \mathtt{S} \rangle$ *be a runtime environment and* Γ_1 *a typing environment where* $\Gamma_1 \approx \Delta_1$*. If* $\Gamma_1 \vdash \mathtt{S} : \Gamma_2$ *for* $\mathtt{S} \notin \{\mathtt{n = e(\overline{e})}, \mathtt{return\ e}\}$ *and* $\langle \Delta_1, \mathtt{S} \rangle \longrightarrow \langle \Delta_2, \mathtt{S'} \rangle$*, then* $\Gamma_2 \approx \Delta_2$.

Proof 15 *By induction on the structure of* $\mathtt{S}$.

- *Case "*$\mathtt{x = e}$*": By T-VASSIGN we have* $\Gamma_2 = \Gamma_1[\mathtt{x} \mapsto \mathtt{T_e}]$*, where* $\Gamma_1 \vdash \mathtt{e} : \mathtt{T_e}$*. Similarly, by R-VASSIGN, we have* $\Delta_2 = \Delta_1[\mathtt{x} \mapsto \mathtt{v}]$ *where* $\Delta_1 \vdash \mathtt{e} \rightsquigarrow \mathtt{v} : \mathtt{T_v}$*. Since* $\Gamma_1 \approx \Delta_1$*, it follows that* $\mathtt{T_v} \leq \mathtt{T_e}$ *by Lemma 9 and, hence, that* $\Gamma_2 \approx \Delta_2$.

- *Case "*$\mathtt{x.f = e}$*": By T-FASSIGN we have* $\Gamma_1 \vdash \mathtt{e} : \mathtt{T_e}$*,* $\Gamma_1 \vdash \mathtt{x} : \mathtt{T_x}$ *and* $\Gamma_2 = \Gamma_1[\mathtt{x} \mapsto \mathtt{T'_x}]$*, where* $\mathtt{T'_x}$ *is formed from* $\mathtt{T_x}$ *by updating field* $\mathtt{f}$ *to* $\mathtt{T_e}$*. Similarly, by R-VASSIGN, we have* $\Delta_2 = \Delta_1[\mathtt{x} \mapsto \mathtt{v'}]$ *where* $\Delta_1 \vdash \mathtt{e} \rightsquigarrow \mathtt{v} : \mathtt{T_v}$*,* $\Delta_1(\mathtt{x}) = \{\overline{\mathtt{n\ :\ v}}\}$ *and* $\mathtt{v'} = \{\overline{\mathtt{n\ :\ v}}\}[\mathtt{f} \mapsto \mathtt{v}]$*. Since* $\mathtt{T_v} \leq \mathtt{T_e}$ *by Lemma 9, we have* $\Gamma_2 \approx \Delta_2$.

- *Case "*$\mathtt{x[e_1] = e_2}$*": By T-LASSIGN we have* $\Gamma_2 = \Gamma_1[\mathtt{x} \mapsto [\mathtt{T_1} \sqcup \mathtt{T_2}]]$*, where* $\Gamma_1 \vdash \mathtt{x} : [\mathtt{T_1}]$ *and* $\Gamma_1 \vdash \mathtt{e_2} : \mathtt{T_2}$*. By R-LASSIGN,* $\Delta_1 \vdash \mathtt{e_1} \rightsquigarrow \mathtt{i} : \mathtt{int}$*,* $\Delta_1 \vdash \mathtt{e_2} \rightsquigarrow \mathtt{v} : \mathtt{T'_2}$*,* $\Delta_1 \vdash \mathtt{x} \rightsquigarrow [\overline{\mathtt{v}}] : [\mathtt{T'_1}]$ *and, since* $\Gamma_1 \approx \Delta_1$*, we have* $\mathtt{T'_1} \leq \mathtt{T_1}$ *and* $\mathtt{T'_2} \leq \mathtt{T_2}$ *by Lemma 9. Furthermore,* $\Delta_2 = \Delta_1[\mathtt{x} \mapsto [\overline{\mathtt{w}}]]$ *where* $\mathtt{w_i} = \mathtt{v}$ *and* $\forall_{\mathtt{j} \neq \mathtt{i}}.[\mathtt{w_j} = \mathtt{v_i}]$*. So,* $\Delta_2 \vdash \mathtt{x} \rightsquigarrow [\overline{\mathtt{w}}] : [\mathtt{T'_1} \sqcup \mathtt{T'_2}]$ *and, thus,* $\Gamma_2 \approx \Delta_2$ *since* $\mathtt{T'_1} \sqcup \mathtt{T'_2} \leq \mathtt{T_1} \sqcup \mathtt{T_2}$.

- *Case "*$\mathtt{if\ v \sim= T\!:\ S_1\ else\!:\ S_2}$*": By T-IF we have* $\Gamma_1 \vdash \mathtt{S_1} : \Gamma_3$ *and* $\Gamma_1 \vdash \mathtt{S_2} : \Gamma_4$*, where* $\Gamma_2 = \Gamma_3 \sqcup \Gamma_4$*. Likewise, we have either* $\langle \Delta_1, \mathtt{S_1} \rangle \longrightarrow \langle \Delta_2, \epsilon \rangle$ *by R-IF1 and* $\Gamma_3 \approx \Delta_2$ *(by the induction hypothesis), or* $\langle \Delta_1, \mathtt{S_2} \rangle \longrightarrow \langle \Delta_2, \epsilon \rangle$ *by R-IF2 and* $\Gamma_4 \approx \Delta_2$*. Thus,* $\Gamma_3 \sqcup \Gamma_4 \approx \Delta_2$ *by Lemma 10 and, hence,* $\Gamma_2 \approx \Delta_2$ *(since* $\Gamma_2 = \Gamma_3 \sqcup \Gamma_4$*).*

- *Case "*$\mathtt{while\ e_1 \leq e_2\!:\ S}$*": In executing this loop, the machine goes through a sequence of zero or more transitions of the form* $\langle \Delta_1, \mathtt{S} \rangle \longrightarrow \langle \Delta'_1, \mathtt{S} \rangle$*,* $\langle \Delta'_1, \mathtt{S} \rangle \longrightarrow \langle \Delta''_1, \mathtt{S} \rangle$*, etc. Considering T-WHILE, we arrive at a typing* $\Gamma_1 \sqcup \Gamma_n \vdash \mathtt{S} : \Gamma_1 \sqcup \Gamma_n$*. Since* $\Gamma_1 \approx \Delta_1$*, we have by Lemma 10 that* $\Gamma_1 \sqcup \Gamma_n \approx \Delta_1$*. Then, by the induction hypothesis, we get that* $\Gamma_1 \sqcup \Gamma_n \approx \Delta'_1$*,* $\Gamma_1 \sqcup \Gamma_n \approx \Delta''_1$ *and so on.*

$\square$

Essentially, we limit our preservation theorem to the *intraprocedural* cases as typing is inherently an intraprocedural activity. A coherence theorem handles the movement between functions:

Definition 21 (Invocation) *Let* $\langle \Delta_1, \mathtt{S} \rangle \Longmapsto \langle \Delta_2, \mathtt{S'} \rangle$ *represent the shortest possible sequence which matches* $\langle \Delta_1, \mathtt{S} \rangle \longrightarrow \ldots \longrightarrow \langle \Delta_2, \mathtt{S'} \rangle$.

Theorem 6 (Coherence) *Let* $\langle \Delta_1, \mathtt{n = e(\overline{e})\ ;\ S} \rangle$ *be a runtime environment and* Γ_1 *a typing environment where* $\Gamma_1 \approx \Delta_1$*. If* $\Gamma_1 \vdash \mathtt{n = e(\overline{e})} : \Gamma_2$ *and* $\langle \Delta_1, \mathtt{n = e(\overline{e})\ ;\ S} \rangle \Longmapsto \langle \Delta_2, \mathtt{S} \rangle$*, then* $\Gamma_2 \approx \Delta_2$.

Proof 16 *By induction on the structure of the induced call graph. Let* k *indicate the number of function invocations occuring in the sequence* $\langle \Delta_1, \mathtt{n = e(\overline{e})\ ;\ S} \rangle \Longmapsto \langle \Delta_2, \mathtt{S} \rangle$*:*

- $k = 0$. *Straightforward as, if no other function invocations occur, then* $\Gamma_2 \approx \Delta_2$ *essentially follows immediately from Theorem 5.*
- $k = n$. *Consider the first such subsequence* $\langle \Delta_3, \mathtt{n}{=}\mathtt{e}(\overline{\mathtt{e}})\ ;\ \mathtt{S}' \rangle \Longmapsto \langle \Delta_4, \mathtt{S}' \rangle$. *If* Γ_3 *and* Γ_4 *are the typing environments before and after this statement, then* $\Gamma_3 \approx \Delta_3$ *by Theorem 5, and* $\Gamma_4 \approx \Delta_4$ *by the induction hypothesis. At this point, we have reduce the problem to the* $k - 1$ *case.*

□

6 Related Work

In this section, we concentrate primarily on work relating to Whiley's flow-sensitive type system.

The Strongtalk system prioneered the use of structural typing for describing structures from an untyped world (i.e. SmallTalk) [61]. It is perhaps the most closely aligned work to Whiley. The essential idea is succinctly captured in the following quote (from [61]):

> *"Smalltalk is an unusually flexible and expressive language. Any type system for Smalltalk should place a high priority on preserving its essential flavor."*

The paper hints that some form of recursive structural typing was supported, although few details are given and no formalisation is included. Unfortunately, structural subtyping was subsequently dropped from Strongtalk [62], in part due to the complexity of error messages produced. Although not the subject of this paper, we believe the quality of error messages in a structural type system can be improved by storing the names associated with types at their declarations. Then, when reporting an error, the compiler looks up the type in question (or those isomorphic to it) and matches any name(s) in the current file, or imported files.

The work of Guha *et al.* focuses on flow-sensitive type checking for JavaScript [63]. 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* [64]. 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 [13]. 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 requires a flow-sensitive type checker [65]. Since locals and stack locations are untyped in Java Bytecode, it must infer their types to ensure type safety. Like Whiley, the verifier updates the type of a variable after an assignment, and combines types at control-flow join points using a least upper bound operator. However, it does not update the type of a variable after an `instanceof` test. Furthermore, 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. [66]).

Type qualifiers constrain the possible values a variable may hold. CQual is a flow-sensitive qualifier inference supporting numerous type qualifiers, including those for synchronisation and file I/O [17]. CQual does not account for the effects of conditionals and, hence, retyping is impossible. Fähndrich and Leino discuss a system for checking non-null qualifiers in the context of C# [34]. Here, variables are annotated with `NonNull` to indicate they cannot hold **`null`**. Non-null qualifiers are interesting because they require variables be retyped after conditionals (i.e. retyping `v` from `Nullable` to `NonNull` after `v!=`**`null`**). Fähndrich and Leino hint at the use of retyping, but focus primarily on issues related to object constructors. Ekman *et al.* implemented this system within the JustAdd compiler, although few details are given regarding variable retyping [35]. Pominville *et al.* also briefly discuss a flow-sensitive non-null analysis built using SOOT, which does retype variables after `!=`**`null`** checks [33]. The JACK tool is similar, but focuses on bytecode verification instead [38]. JavaCOP provides an expressive language for writing type system extensions, including non-null types [43]. This system is flow-insensitive and cannot account for the effects of conditionals; as a work around, the tool allows assignment from a nullable variable `x` to a non-null variable if this is the first statement after a `x!=null` conditional.

Information Flow Analysis is the problem of tracking the flow of information, usually to restrict certain flows for security reasons. The work of Hunt and Sands is relevant here, since they adopt a flow-sensitive approach [18]. Their system is presented in the context of a simple While language not dissimilar to ours, although they do not account for the effect of conditionals. Russo *et al.* use an extended version of this system to compare dynamic and static approaches [67]. 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 [68]. Finally, Chugh *et al.* developed a constraint-based (flow-insensitive) information flow analysis of JavaScript [69].

Typestate Analysis focuses on flow-sensitive reasoning about the state of objects, normally to enforce temporal safety properties. Typestates are finite-state automatons which can encode usage rules for common APIs (e.g. a file is never read before being opened), and were pioneered by Strom and Yellin [70, 71]. Fink *et al.* present an interprocedural, flow-sensitive typestate verification system which is staged to reduce overhead [72]. Bodden *et al.* develop an interprocedural typestate analysis which is flow-sensitive at the intra-procedural level [73]. This is a hybrid system which attempts to eliminate all failure points statically, but uses dynamic checks when necessary. This was later extended to include a backward propagation step that improves precision [74].

6.1 Type Inference

Type inference is the process of inferring a type for each variable in an untyped (or partially untyped) program. This is generally done by generating and solving either unification- (e.g. [75]) or set-constraints (e.g. [76, 46, 77]) over the program in question.

Numerous type inference systems have been developed for object-oriented languages (e.g. [76, 78, 46, 77, 15, 16]). These, almost exclusively, assume the original program is completely untyped and employ set constraints (see [79]) as the mechanism for inferring types. As such, they address a 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*).

Gagnon *et al.* present a technique for converting Java Bytecode into an intermediate representation with a single static type for each variable [80]. 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 [9]. This uses a very different technique from us, known as *bidirectional type checking*, which was first developed for System F by Pierce and Turner [81]. This approach is suitable for C# 3.0 which does not permit variables to have different types at different program points.

7 Conclusion

We have presented the flow-sensitive and structural type system developed for the Whiley language. This permits variables to be declared implicitly, have multiple types within a function, and be retyped after runtime type tests. The semantics of Whiley also allows flexible typing of updates to compound structures. Additionally, subtyping between user-defined types is implicit. The result is a statically-typed language which, for the most part, has the look and feel of a dynamic language. An open source implementation of Whiley is available from `http://whiley.org`.

We formalised the type system using a core calculus called Featherweight Whiley (FW). We provided both a semantic and algorithmic interpretation of types including a coinductively defined subtype relation. The typing algorithm requires a distinct typing environment at each program point (much like flow-sensitive program analysis). Typing environments are merged at control-flow joins, whilst a greatest lower bound (resp. greatest difference) operator is used for retyping variables on the true (resp. false) branches of runtime type tests. We have sketched soundness and termination properties for this system — Full details can be found in [82].

Finally, there are several improvements to Whiley's type system that could be considered in the future. Most notably, the absence of generic types (i.e. type polymorphism) can result in the need for additional runtime type tests, compared with dynamically typed languages. The greatest difference operator, as discussed in §3.5, often produces conservative results. This could be resolved by including negation and intersection types [46, 83].

Acknowledgements. The authors would especially like to thank Neal Glew, Nick Cameron, Roman Klapaukh, Art Protin and Lindsay Groves for helpful comments on earlier drafts of this paper.

References

[1] R. Cartwright and M. Fagan. Soft typing. In *Proc. PLDI*, pages 278–292. ACM Press, 1991.

[2] 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.

[3] John K. Ousterhout. Scripting: Higher-level programming for the 21st century. *IEEE Computer*, 31(3):23–30, 1998.

[4] Diomidis Spinellis. Java makes scripting languages irrelevant? *IEEE Software*, 22(3):70–71, 2005.

[5] Ronald Prescott Loui. In praise of scripting: Real programming pragmatism. *IEEE Computer*, 41(7):22–26, 2008.

[6] B. Bloom, J. Field, N. Nystrom, J. Östlund, G. Richards, R. Strnisa, J. Vitek, and T. Wrigstad. Thorn: robust, concurrent, extensible scripting on the JVM. In *Proc. OOPSLA*, pages 117–136, 2009.

[7] Linda Dailey Paulson. Developers shift to dynamic programming languages. *IEEE Computer*, 40(2):12–15, 2007.

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

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

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

[11] J. Siek and W. Taha. Gradual typing for objects. In *Proc. ECOOP*, volume 4609 of *LNCS*, pages 151–175. Springer-Verlag, 2007.

[12] T. Wrigstad, F. Nardelli, S. Lebresne, J. Östlund, and J. Vitek. Integrating typed and untyped code in a scripting language. In *Proc. POPL*, pages 377–388, 2010.

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

[14] Cormac Flanagan. Hybrid type checking. In *Proc. POPL*, pages 245–256. ACM Press, 2006.

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

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

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

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

[19] 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.

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

[21] D. Malayeri and J. Aldrich. Integrating nominal and structural subtyping. In *Proc. ECOOP*, pages 260–284, 2008.

[22] Luca Cardelli. Structural subtyping and the notion of power type. In *Proc. POPL*, pages 70–79. ACM Press, 1988.

[23] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

[24] Nurudeen Lameed and Laurie J. Hendren. Staged static techniques to efficiently implement array copy semantics in a MATLAB JIT compiler. In *Proc. CC*, volume 6601 of *LNCS*, pages 22–41. Springer-Verlag, 2011.

[25] Natarajan Shankar. Static analysis for safe destructive updates in a functional language. In *In Proc. LOPSTR*, pages 1–24, 2001.

[26] Martin Odersky. How to make destructive updates less destructive. In *Proc. POPL*, pages 25–36, 1991.

[27] Paul F. Dietz and Daniel D. Sleator. Two algorithms for maintaining order in a list. In *Proc. STOC*, pages 365–372. ACM Press, 1987.

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

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

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

[31] Tony Hoare. Null references: The billion dollar mistake, presentation at qcon, 2009.

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

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

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

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

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

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

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

[39] Laurent Hubert. A non-null annotation inferencer for java bytecode. In *Proc. PASTE*, pages 36–42. ACM, 2008.

[40] L. Hubert, T. Jensen, and D. Pichardie. Semantic foundations and inference of non-null annotations. In *Proc. FMOODS*, pages 132–149, 2008.

[41] F Barbanera and M Dezani-CianCaglini. Intersection and union types. In *Proc. of TACS*, volume 526 of *LNCS*, pages 651–674, 1991.

[42] Atsushi Igarashi and Hideshi Nagira. Union types for object-oriented programming. *JOT*, 6(2), 2007.

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

[44] ISO/IEC. international standard ISO/IEC 9899, programming languages — C, 1990.

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

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

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

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

[49] J. E. Hopcroft. An $n \log n$ algorithm for minimizing the states in a finite-automaton. In *TMC*, pages 189–196. 1971.

[50] Bruce W. Watson and Jan Daciuk. An efficient incremental dfa minimization algorithm. *NLE*, 9:49–64, 2003.

[51] Marco Almeida, Nelma Moreira, and Rogério Reis. Incremental DFA minimisation. In *CIAA*, pages 39–48, 2010.

[52] J. Hopcroft, R. Motwani, and J. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2006.

[53] B. D. McKay. Practical graph isomorphism. In *NMC*, volume 30, pages 45–87, 1981.

[54] Babai and Luks. Canonical labeling of graphs. In *Proc. STOC*, pages 171–183, 1983.

[55] T. Miyazaki. The complexity of mckay's canonical labelling algorithm. *Groups and Computation, II*, 28:239–256, 1997.

[56] J. Faulon. Isomorphism, automorphism partitioning, and canonical labeling can be solved in polynomial-time for molecular graphs. *JCICS*, 38(3):432–444, 1998.

[57] François Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Proceedings of the conference on Formal Methods in Programming and their Applications*, pages 128–141. Springer-Verlag, 1993.

[58] Patrick Cousot and Radhia Cousot. Comparison of the Galois connection and widening/narrowing approaches to abstract interpretation. *BIGRE*, 74:107–110, 1991.

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

[60] Xavier Leroy and Hervé Grall. Coinductive big-step operational semantics. *Information and Computation*, 207(2):284–304, 2009.

[61] G. Bracha and D. Griswold. Strongtalk: Typechecking smalltalk in a production environment. In *OOPSLA*, pages 215–230, 1993.

[62] Gilad Bracha. The strongtalk type system for smalltalk.

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

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

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

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

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

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

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

[70] R. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE TSE*, 12(1):157–171, 1986.

[71] Robert E. Strom and Daniel M. Yellin. Extending typestate checking using conditional liveness analysis. *IEEE TSE*, 19(5):478–485, 1993.

[72] S. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. *ACM TOSEM*, 17(2):1–9, 2008.

[73] Eric Bodden, Patrick Lam, and Laurie J. Hendren. Finding programming errors earlier by evaluating runtime monitors ahead-of-time. In *Proc. ESEC/FSE*, pages 36–47. ACM Press, 2008.

[74] E. Bodden. Efficient hybrid typestate analysis by determining continuation-equivalent states. In *Proc. ICSE*, pages 5–14, 2010.

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

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

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

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

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

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

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

[82] D. J. Pearce and J. Noble. Flow-sensitive types for whiley. Technical Report ECSTR10-23, Victoria University of Wellington, 2010.

[83] A. Frisch, G. Castagna, and V. Benzaken. Semantic subtyping. In *Proc. LICS*, pages 137–146. IEEE Computer Society Press, 2002.