Whiley: a Language Combining Flow-Typing with Updateable Value Semantics

David J. Pearce, Nicholas Cameron and James Noble

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

April 2012

Abstract

Whiley is a statically typed language with the look-and-feel of a dynamically typed language. For example, as with many dynamically typed languages, assignments to variables, fields and list elements always succeed. Key to this is a careful combination of *flow typing* and *updateable value semantics*. This enables a programming model that lies between the imperative (where mutable data is passed by reference and updated in place) and the functional (where immutable data is passed by value or by name). In Whiley, mutable data that may be updated in place is passed and returned by value, with the compiler free to eliminate unnecessary copies. Combined with flow typing (where variables may have different types at different program points), this yields an unusually flexible type system.

In this paper, we explore the advantages of a programming model built around flow typing and updateable value semantics. Building on our previous work, we develop an extended flow typing calculus capturing the salient features of Whiley. Soundness and termination is proved. Finally, to address concerns that Whiley's programming model cannot be made efficient, we report on experiments demonstrating that simple optimisations eliminate many unnecessary copies, and that the vast majority of update operations can be performed in place.

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 [3–6]. A common complaint against statically typed languages is the need for often unnecessarily verbose type declarations. Hindley-Milner Type inference [7, 8] is a common approach to addressing this problem, where type declarations are inferred automatically. Scala [9], C#3.0 [10] and OCaml [11] provide good examples of this in an imperative setting. However, these languages fall short of the flexibility offered by a dynamically typed language. This is because they still require each program variable to have exactly one type.

Consider this JavaScript example:

```
x = { f: 1 };   // point 1
document.write(x.f);
x.f = "ONE";    // point 2
x.g = "ONE";    // point 3
document.write(x.f + "␣" + x.g);
```

Typing this code fragment requires a type system where program variables can have different types at different program points. For example, $\mathtt{x}^1 = \{\mathtt{int\ f}\}$, $\mathtt{x}^2 = \{\mathtt{string\ f}\}$ and $\mathtt{x}^3 = \{\mathtt{string\ f},\ \mathtt{string\ g}\}$ is one possible typing of the above, where $\mathtt{x}^\ell$ is the type of x at point ℓ.

1.1 Flow Typing

Flow typing offers an alternative to Hindley-Milner type inference where a variable may have different types at different program points. The technique is adopted from flow-sensitive program analysis and has been used for non-null types [12–15], information flow [16–18], purity checking [19] and more [12, 13, 20–25]. Groovy 2.0

A defining characteristic of flow typing is the ability to *retype* a variable — that is, assign it a completely unrelated type. The JVM Bytecode Verifier [26], perhaps the most widely-used example of a flow typing system, provides a good illustration:

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

In the above, register 0 contains the parameter value on entry and, initially, has type **int**. The type of register 0 is subsequently changed to `float` by the `fstore` bytecode. To ensure type safety, the JVM bytecode verifier employs a typing algorithm based upon dataflow analysis [27]. This tracks the type of a variable at each program point, allowing it easily to handle the above example.

Flow typing can also retype variables after conditionals. A *non-null type system* (e.g. [12–15]) prevents variables which may hold **null** from being dereferenced. The following illustrates:

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

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

1.2 Whiley

Whiley [28–30] employs a flow type system to give it the look-and-feel of a dynamically typed language. Underpinning this is a model based on *updateable value semantics*, without which many of the benefits from flow typing would be unsafe. To see why, consider this Whiley program:

```
define iPoint as {int x, int y}
define rPoint as {real x, real y}

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

Here, the type of `p` is updated from `{int x, int y}` to `{real x,int y}` after `p.x` is assigned, and then to `{real x,real y}` after `p.y` is assigned.

Suppose that variable `p` was a *reference* to an `iPoint` above (as it would be in e.g. Java). Then, the above is unsafe because other aliases may exist for the object to which `p` refers. If these were created before `normalise()` was called, they would continue to assume (incorrectly) that fields `x` and `y` contain values of type **int**.

In fact, the above Whiley program is safe. This is because, in Whiley, both primitive data types (arbitrary-precision integers and rationals) and compound data types (lists, records, sets, and maps) have *value semantics*. This means they are passed and returned by value (as in Pascal's value parameters, MATLAB or most functional languages). But, unlike functional languages (and like Pascal), values of compound types can be updated in place. For example, when a list is passed as a parameter to a function, the list is effectively cloned. Within a function body, the list can be updated in place and, with updateable value semantics, no other list is affected. That is, assigning to a list effectively creates a new list with the updated

element. The only communication a function has with its calling context is via the result it returns, which again is passed by value.

On the surface, value semantics may appear somewhat inefficient — however, a variety of techniques exist to ensure copying is only done when absolutely necessary [31–33]. Furthermore, as is well known, there are other advantages to having value semantics: it simplifies hard problems, such as verification [34, 35] and compiler optimisation [36, 37].

1.3 Contributions

We believe the particular point that Whiley represents in the design space — namely, flow typing with updateable value semantics — has been under-explored, and we seek to redress this imbalance. The contributions of this paper are:

1. We present a novel type system which combines structural and flow typing with updateable value semantics. This is in the context of Whiley, a statically typed language which compiles to the JVM.

2. We formalise Whiley's type system using a small calculus, called Featherweight Whiley (FW). This employs a *constraint-based* formulation of the typing rules and builds on our previous work formalising a minimal flow type system called FT [38, 39]. Specifically, FW extends FT to include *effective unions*, *negations* and *type tests*. We include proofs of termination and soundness for FW.

3. We report on experimental results investigating the impact of updateable value semantics. The conclusion is that, although the semantics of Whiley dictates that compound structures must appear to be copied, in practice they almost never are.

An open source implementation of Whiley is freely available from `http://whiley.org` and details of Whiley's JVM implementation can be found here [30].

2 Language Overview

Through a series of small examples, we now examine Whiley's flow typing system in more detail. In particular, we explore its benefits and how these are facilitated by the updateable value semantics model. Our aim is not to provide a comprehensive language reference up front but, instead, to progressively introduce the interesting features.

Example 1 — Union Types. Nullable references have proved a significant source of error in e.g. Java [40]. The issue is that one can treat *nullable* references as though they are *non-null* references [41], and many solutions have been proposed (e.g. [12–15, 21, 42–45]). Whiley's flow type system lends itself naturally to handling this problem through *union types* (e.g. [46, 47]). 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 this way, Whiley's flow 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`**".

Example 2 — Structural Typing. Statically typed languages, such as Java, employ nominal typing of data structures (e.g. `class`es and `interface`s), coupled with explicit programmer-defined subtyping (e.g. `extends` and `implements`). Whilst this can result in rigid and difficult-to-extend hierarchies (see e.g. [48]), it also prohibits some benefits from flow typing. In contrast, Whiley employs *structural typing* [49] for greater flexibility and to help fully utilise the flow typing system. Consider the following (simplified) excerpt from a chess game validator:

```
define Pos as { int col, int row }
define Move as { Piece piece, Pos from, Pos to }

define ShortPos as null|{int row}|{int col}
define ShortMove as { Piece piece, ShortPos from, Pos to }

[Pos] find(Piece p, Board b):
    ...

[Pos] narrow(ShortMove s, [Pos] ms, Board b):
    ...

Move convert(ShortMove m, Board b):
    matches = find(m.piece,b)
    matches = narrow(m,matches,b)
    if |matches| != 1:
        throw Error("invalid move")
    m.from = matches[0]
    return m
```

The chess benchmark checks the validity of games given in short algebraic notation. Here, moves are given in an abbreviated (and potentially ambiguous) form with only the destination square given. For

example, a move "Nf6" indicates a Knight moving to square "f6". If the player has only one knight, then the move must refer to it. However, if there are two Knights, the system must determine which it is. This is done by `narrow()` above, which intersects the destination with the possible moves of the matching pieces. In the case of multiple matches, an `Error` is thrown. Short Algebraic Notation also permits explicit disambiguation by providing either the *rank* or *file* of the given piece. For example, "Neg3" indicates the Knight on file "e" moves to square "g3". Hence, `ShortPos` is either **`null`** (indicating no position given) or gives the `rank` or `file` for disambiguation.

Structural and flow typing come together when typing the above fragment. The critical issue resolves around the statement "`m.from = matches[0]`". This *retypes* variable `m` from `ShortMove` to `Move`, allowing the subsequent statement "**`return`** `m`" to be type checked. Since this retyping is done by structural assignment, it requires a structural notion of subtyping between `ShortMove` and `Move`.

Example 3 — Effective Unions Types. A union of types of the same kind (e.g. a union of record types, or a union of list types) can expose commonality and are called *effective unions* (e.g. an effective record type). In the case of a union of records, fields common to all records are exposed. The following (simplified) excerpt, taken from a PNG image decoder, illustrates:

```
define IHDR_TYPE as 0x52444849
define IHDR as {int type,int width,int height}
define IDAT as {int type,[byte] data}
define CHUNK as IHDR | IDAT

[CHUNK] decode([byte] data):
   chunks = []
   pos = 0
   // parse chunks
   while pos < |data|:
      chunk,pos = parse(data,pos)
      chunks = chunks + [chunk]
   // check first chunk
   assert |chunks| > 0
        && chunks[0].type == IHDR_TYPE
   // done
   return chunks

CHUNK parse([byte] data, int pos):
   ...
```

A PNG file defines a list of `CHUNK`s, where the first `CHUNK` must be an instance of `IHDR` [50]. The above simply decodes the bytes of a PNG file into the list of `CHUNK`s. It also checks the first `CHUNK` is an `IHDR`.

In the above, the expression `chunks[0]` has union type `IHDR|IDAT` (a.k.a type `CHUNK`). The record types `IHDR` and `IDAT` differ and are not related through subtyping. However, they share a common field, namely `type`. Thus, `IHDR|IDAT` defines an *effective record type* of `{int type, ...}` which means `chunks[0].type` is considered type safe. Finally, it's interesting to note that the notion of an effective record type is similar, in some ways, to that of the *common initial sequence* found in C [51].

Example 4 — Recursive Data Types. To represent abstract data types, Whiley provides recursive types which are, in many ways, similar to the abstract data types found in functional languages (e.g. Haskell, ML, etc). One difference is that they are *structurally* rather than *nominally* typed. For example:

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

int length(LinkedList l):
  if l is null:
    return 0 // l now has type null
  else:
    return 1 + length(l.next) // l now has type {int data, LinkedList next}
```

Here, we again see how flow typing gives an elegant solution. More specifically, on the false branch of the type test "`l` **`is null`**", variable `l` is automatically retyped to `{int data, LinkedList next}` —

thus ensuring the subsequent dereference of `l.next` is safe. No casts are required as would be needed for a conventional imperative language (e.g. Java). Finally, like all compound structures, the semantics of Whiley dictates that recursive data types are passed by value (or, at least, appear to be from the programmer's perspective).

Example 5 — Negation Types. Retyping variables after type tests necessitates the use of *negation types* (see e.g. [52]). A negation type `!T` defines the set of values *not* in `T`. The following example (albeit artificial) illustrates:

```
int f(any x):
    if x is {any field}:
        return 0
    else if x is {int field}:
        return 1
    else:
        return 2
```

The above code does not type check. This is because the type of `x` on the false branch of the first condition is `!{any field}`. Furthermore, `{int field}` is a subtype of `{any field}` and, hence, *not* a subtype of `!{any field}`. Therefore, the compiler emits an error message stating that "`return 1`" is dead-code.

Example 6 — List Retyping. The ability to retype lists is essential to their operation in Whiley. That's because, to give the look-and-feel of a dynamically typed language, Whiley does not require local variables be declared. The following excerpt from an implementation of the DEFLATE decompression algorithm [53] gives an unusual illustration:

```
codeLengths = []

for i in 0..len:
  clen,reader = BitBuffer.readUInt(reader,3)
  j = lengthCodeMap[i]
  while |codeLengths| <= j:
    codeLengths = codeLengths + [0]
  codeLengths[j] = clen
```

Here, `lengthCodeMap` is a list of **int**s. Thus, `j` has type **int** and the DEFLATE format ensures that `codeLengths[j]` is never out of bounds.

In the above, an import question is what type(s) are given to `codeLengths`. After the initial assignment, `codeLengths` has type `[void]`. Within the **for**-loop, `codeLengths` is appended with an **int**, and has an element assigned the value of `clen` (also an **int**). Therefore, within (and after) the **for**-loop, `codeLengths` has type `[int]`. Since `[void]` is a subtype of `[int]`, this fragment is considered type safe.

The above example relies on the type of `codeLengths` being able to seamlessly change from `[void]` to `[int]`. One might wonder, however, whether the compiler could initially guess an appropriate type to avoid retyping. Unfortunately, in the general case, one cannot avoid such retypings. The following (somewhat contrived) example illustrates:

```
[int|null] f([int] xs, int target):
  for i in 0..|xs|:
    if xs[i] == target:
      xs[i] = null
  return xs
```

Here, an element of the list `xs` is **null**ed out. To avoid relying on an interprocedural type checking algorithm (which would likely be prohibitively expensive), Whiley requires parameters and returns to have declared types. Thus, the variable `xs` must be retyped by the **for**-loop as a result of the assignment "`xs[i] = null`". Finally, recall that the retyping employed in this example is safe because compound data types (lists, records, sets, and maps) have *value semantics*. That is, they are passed and returned by value and, furthermore, when e.g. a list is updated it is effectively cloned. Hence, `xs` is not a *reference* to a list (as in e.g. Java); rather, it *is* a list. This protects against the unsoundness that such retyping would present in most object-oriented languages (e.g. Java).

2.1 Exceptions

Introduce exceptions, and talk about how the scoping issue in Java is resolved.

```
int g() throws Error:
    return 1

int f():
    try:
        x = g()
    catch(Error e):
        return 0
    return x        // safe
```

2.2 Efficiency

The use of value semantics may seem inefficient, as it implies much unnecessary copying of data. However, techniques exist (e.g. reference counting) to ensure data is copied only when absolutely necessary [31–33]. Whiley employs reference counting of compound structures to increase efficiency, as we discuss in §5.

In languages like Java, programmers often also forgo the advantages of passing data by reference, preferring *defensive copying* instead [54]. This is particularly prevalent for constructors, as highlighted by the following from Item 24 of Effective Java [54]:

> "*It is essential to make a defensive copy of each mutable parameter to the constructor*"

The argument here is that, to avoid unexpected side-effects from external sources, an object should *own* all of its components. Whether or not this recommendation is widely adopted in practice remains uncertain. Nevertheless, it hints at ways in which value semantics could actually increase efficiency. This is because many defensive copies may not, in fact, be necessary — but the programmer has no way to know this ahead of time. With value semantics and reference counting, as in Whiley, such copies are automatically eliminated.

3 Featherweight Whiley

We now introduce our calculus which, in the spirit of Featherweight Java [55], is called *Featherweight Whiley (FW)*. The calculus is specifically kept to a minimum to allow us to succinctly capture the main aspects of Whiley. In this section, we introduce the syntax, semantics and subtyping rules for FW.

3.1 Types

The following gives a *syntactic* definition of types in FW:

$$\mathtt{T} ::= \mathtt{any} \mid \mathtt{int} \mid \{\mathtt{T}_1\ \mathtt{f}_1, \ldots, \mathtt{T}_n\ \mathtt{f}_n\} \mid \neg\mathtt{T} \mid \mathtt{T}_1 \wedge \ldots \wedge \mathtt{T}_n \mid \mathtt{T}_1 \vee \ldots \vee \mathtt{T}_n$$

Here, **any** represents $\top$, **int** the set of all integers and $\{\mathtt{T}_1\ \mathtt{f}_1, \ldots, \mathtt{T}_n\ \mathtt{f}_n\}$ represents records with one or more fields. The union $\mathtt{T}_1 \vee \mathtt{T}_2$ is a type whose values are in $\mathtt{T}_1$ *or* $\mathtt{T}_2$. Union types are generally useful in flow typing systems, as they can characterise types generated at meet points in the control-flow graph. The intersection $\mathtt{T}_1 \wedge \mathtt{T}_2$ is a type whose values are in $\mathtt{T}_1$ *and* $\mathtt{T}_2$. Intersections are needed in our flow type system to capture the type of a variable (e.g. `x`) after a type test (e.g. `x is T`). The type $\neg\mathtt{T}$ is the *negation* type containing those values *not* in `T`. Negations are useful for capturing the type of a variable on the false branch of a type test.

To better understand the meaning of types in FW, it is helpful to give a *semantic interpretation* (following e.g. [52, 56–58]). The aim is to give a set-theoretic model where subtype corresponds to subset. The *domain* $\mathbb{D}$ of values in our model consists of the integers and all records constructible from values in $\mathbb{D}$:

$$\mathbb{D} = \mathbb{Z} \cup \left\{ \{\mathtt{f}_1 : \mathtt{v}_1, \ldots, \mathtt{f}_n : \mathtt{v}_n\} \mid \mathtt{v}_1 \in \mathbb{D}, \ldots, \mathtt{v}_n \in \mathbb{D} \right\}$$

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

$$\begin{aligned}
[\![\mathtt{any}]\!] &= \mathbb{D} \\
[\![\mathtt{int}]\!] &= \mathbb{Z} \\
[\![\{\mathtt{T}_1\ \mathtt{f}_1, \ldots, \mathtt{T}_n\ \mathtt{f}_n\}]\!] &= \{\mathtt{f}_1 : \mathtt{v}_1, \ldots, \mathtt{f}_n : \mathtt{v}_n\}\ \textbf{\textit{forall}}\ \mathtt{v}_1 \in [\![\mathtt{T}_1]\!], \ldots, \mathtt{v}_n \in [\![\mathtt{T}_n]\!] \\
[\![\neg\mathtt{T}]\!] &= \mathbb{D} - [\![\mathtt{T}]\!] \\
[\![\mathtt{T}_1 \wedge \ldots \wedge \mathtt{T}_n]\!] &= [\![\mathtt{T}_1]\!] \cap \ldots \cap [\![\mathtt{T}_n]\!] \\
[\![\mathtt{T}_1 \vee \ldots \vee \mathtt{T}_n]\!] &= [\![\mathtt{T}_1]\!] \cup \ldots \cup [\![\mathtt{T}_n]\!]
\end{aligned}$$

It is important to distinguish *syntactic* representation from the *semantic* model of types. The form corresponds to a physical machine representation, whist the former is a mathematical ideal. As such, the syntactic representation diverges from the semantic model and, to compensate, we must establish a correlation between them. For example `int` and $\neg\neg$`int` have distinct syntactic representations, but are semantically indistinguishable. Similarly for $\{\mathtt{int} \vee \{\mathtt{int\ x}\}\ \mathtt{f}\}$ and $\{\mathtt{int\ f}\} \vee \{\{\mathtt{int\ x}\}\ \mathtt{f}\}$.

Ultimately, we want to construct a subtyping algorithm that is both *sound* and *complete* (i.e. that $\mathtt{T}_1 \leq \mathtt{T}_2 \iff [\![\mathtt{T}_1]\!] \subseteq [\![\mathtt{T}_2]\!]$). The distinction between syntactic and semantic forms presents a significant challenge in doing this. Since this problem is orthogonal to the main contribution of this paper, we simply ignore it for now and tacitly assume that such a subtype algorithm exists.

3.2 Syntax

Figure 1 gives the syntax of FW where $[\![\cdot]\!]^{\ell}$ is not part of the syntax but (following [59]) identifies the distinct program points and associates each with a unique label ℓ. For simplicity, indentation is not explicitly described in the syntax, but is instead assumed. The overbar (as in e.g. $\overline{\mathtt{T\ n}}$) is taken to indicate a list with appropriate separator(s). An example FW program is given below:

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

Here, we see how each distinct program point has a unique label. These are provided to help formalise the typing algorithm, which is discussed later in §4.

Finally, whilst it is clear that FW programs are fairly limited, they can still characterise a number of important issues. Furthermore, it is relatively easy to add additional constructs such as function invocation, arithmetic, etc.

3.3 Semantics

A small-step operational semantics for FW is given in Figure 2. The semantics describe an abstract machine executing statements of the program and (hopefully) halting to produce a value. Here, Δ is the *runtime environment*, whilst `v` denotes *runtime values*. A runtime environment Δ maps variables to their current runtime value.

In Figure 2, `halt(v)` is used to indicate the machine has halted producing value `v`. This must be distinguished from the notion of being "stuck". The latter occurs when the machine has not halted, but cannot execute further (because none of the transition rules from Figure 2 applies). For example, a statement $\texttt{n} = \texttt{m.f}$ can result in the machine being stuck. To see why, notice that only rule R-VF can be applied to such a statement. This has an explicit requirement that `m` currently holds a record value containing at least field `f`. Thus, in the case that `m` does not currently hold a record value, or that it holds a record value which does not contain a field `f`, then the machine will be stuck.

A few simple observations can be made from Figure 2. Firstly, *variables do not need to be explicitly declared* — rather, they are declared implicitly by assignment. Secondly, *variables must be defined before being used* — as, otherwise, the machine will get stuck. Finally, *assignments to fields always succeed*. This is captured in rule R-FV, where the record value being assigned is updated with a (potentially new) field `f`. The following illustrates:

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

This program executes under the rules of Figure 2 without getting stuck. Furthermore, as we will see, it can be type checked with appropriate flow typing rules (§4). The key to this is that variable `x` has different types at different program points: after initialisation, it has type $\{\texttt{int f}\}$; after the subsequent assignment to field `f` this becomes $\{\texttt{any f}\}$; and, finally, after the assignment to field `g` it has type $\{\texttt{any f}, \texttt{int g}\}$.

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

Syntax:

$$
\begin{array}{lll}
\texttt{F} & ::= & \texttt{T f}(\overline{\texttt{T n}})\texttt{ : B} \\
\texttt{B} & ::= & \texttt{S B} \mid \epsilon \\
\texttt{S} & ::= & \llbracket \texttt{n} = \texttt{v} \rrbracket^{\ell} \mid \llbracket \texttt{n} = \texttt{m} \rrbracket^{\ell} \mid \llbracket \texttt{n.f} = \texttt{m} \rrbracket^{\ell} \mid \llbracket \texttt{n} = \texttt{m.f} \rrbracket^{\ell} \mid \llbracket \texttt{return n} \rrbracket^{\ell} \\
 & & \mid \texttt{if } \llbracket \texttt{n is T} \rrbracket^{\ell} \texttt{ : } \texttt{B}_1 \texttt{ else : } \texttt{B}_2 \mid \texttt{while } \llbracket \texttt{n} < \texttt{m} \rrbracket^{\ell} \texttt{ : B} \\
\texttt{v} & ::= & \{\ \overline{\texttt{f : v}}\ \} \mid \mathcal{I}
\end{array}
$$

Figure 1: Syntax for FW. Here, $\texttt{n}, \texttt{m}$ and $\texttt{i}$ represent variable identifiers, whilst $\mathcal{I}$ represents the set of integers.

Semantics:

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

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

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

$$\frac{\Delta(\texttt{n}) = \{\overline{\texttt{f : v}}\} \quad \texttt{v} = \{\overline{\texttt{f : v}}\}[\texttt{f} \mapsto \Delta(\texttt{m})]}{\langle \Delta, \llbracket \texttt{n.f} = \texttt{m} \rrbracket^{\ell}\ \texttt{B} \rangle \longrightarrow \langle \Delta[\texttt{n} \mapsto \texttt{v}],\ \texttt{B} \rangle} \quad \text{(R-FV)}$$

$$\frac{\texttt{v} = \Delta(\texttt{n})}{\langle \Delta, \llbracket \texttt{return n} \rrbracket^{\ell}\ \texttt{B} \rangle \longrightarrow \texttt{halt}(\texttt{v})} \quad \text{(R-RV)}$$

$$\frac{\Delta(\texttt{n}) = \texttt{v} \quad \vdash \texttt{v} : \texttt{T}_1 \quad \texttt{T}_1 \leq \texttt{T}_2}{\langle \Delta, \texttt{if } \llbracket \texttt{n is T}_2 \rrbracket^{\ell} \texttt{: } \texttt{B}_1 \texttt{ else: } \texttt{B}_2 \texttt{ ; } \texttt{B}_3 \rangle \longrightarrow \langle \Delta,\ \texttt{B}_1\ \texttt{B}_3 \rangle} \quad \text{(R-I1)}$$

$$\frac{\Delta(\texttt{n}) = \texttt{v} \quad \vdash \texttt{v} : \texttt{T}_1 \quad \texttt{T}_1 \not\leq \texttt{T}_2}{\langle \Delta, \texttt{if } \llbracket \texttt{n is T}_2 \rrbracket^{\ell} \texttt{: } \texttt{B}_1 \texttt{ else: } \texttt{B}_2 \texttt{ ; } \texttt{B}_3 \rangle \longrightarrow \langle \Delta,\ \texttt{B}_2\ \texttt{B}_3 \rangle} \quad \text{(R-I2)}$$

$$\frac{\Delta(\texttt{n}) < \Delta(\texttt{m})}{\langle \Delta, \texttt{while } \llbracket \texttt{n} < \texttt{m} \rrbracket^{\ell} \texttt{: } \texttt{B}_1 \texttt{ ; } \texttt{B}_2 \rangle \longrightarrow \langle \Delta,\ \texttt{B}_1 \texttt{ while } \llbracket \texttt{n} < \texttt{m} \rrbracket^{\ell} \texttt{: } \texttt{B}_1 \texttt{ ; } \texttt{B}_2 \rangle} \quad \text{(R-W1)}$$

$$\frac{\Delta(\texttt{n}) \geq \Delta(\texttt{m})}{\langle \Delta, \texttt{while } \llbracket \texttt{n} < \texttt{m} \rrbracket^{\ell} \texttt{: } \texttt{B}_1 \texttt{ ; } \texttt{B}_2 \rangle \longrightarrow \langle \Delta,\ \texttt{B}_2 \rangle} \quad \text{(R-W2)}$$

Figure 2: Small-step operational semantics for statements in FW. The semi-colon is used to demarcate the end of a syntactic block as would be indicated by indentation.

4 Flow Type System

We now present a *constraint-based* formulation of the the typing rules for FW in the style of e.g. [56, 60–64]. This approach is less natural than the alternative dataflow formulation (see e.g. [38, 39]). However, there is an important advantage: *typing always terminates in finite time!* More specifically, we employ the following language of type constraints:

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

Here, $\mathtt{T}$ represents a fixed type from those outlined in §3, whilst $\mathtt{n}_\ell$ denotes the set of *labelled* type variables which range over types (though, for simplicity, we will sometimes omit the label). The idea is that, for a given FW program, we generate a set of such constraints and subsequently solve them. The following illustrates the idea:

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

Here, we see that the life of each program variable may be split across multiple constraint variables (e.g. `r` is represented by $\mathtt{r}_1, \mathtt{r}_2$ and $\mathtt{r}_3$). Those familiar with *Static Single Assignment Form* [65–67] will notice a strong similarity. As another example, let us consider how variables are retyped through conditionals:

```
int f(int ∨ {int f} x):                // x0 ≡ int ∨ {int f}
    if x is int[1]:                    // x1+ ≡ x0 ⊓ int
        return x[2]                    // int ⊒ x1+
    else:                              // x1− ≡ x0 ⊓ ¬int
        x = x.f[3]                     // x2 ≡ x1−.f
        return x[3]                    // int ⊒ x2
```

In the above, we see that the type of `x` on the true and false branches is represented (respectively) by $\mathtt{x}_{1+}$ and $\mathtt{x}_{1-}$. The above program type checks because (as we'll see) $(\mathtt{int} \vee \{\mathtt{int\ f}\}) \sqcap \mathtt{int} \implies (\mathtt{int} \vee \{\mathtt{int\ f}\}) \wedge \mathtt{int}$, which gives `int` (for $\mathtt{x}_{1+}$). Likewise, we have $(\mathtt{int} \vee \{\mathtt{int\ f}\}) \sqcap \neg\mathtt{int} \implies (\mathtt{int} \vee \{\mathtt{int\ f}\}) \wedge \neg\mathtt{int}$ for the false branch, giving $\{\mathtt{int\ f}\}$ (for $\mathtt{x}_{1-}$).

Definition 2 (Typing) *A typing, Σ, maps variables to types and* satisfies *a constraint set $\mathcal{C}$, denoted $\Sigma \models \mathcal{C}$, if for all $\mathtt{T} \sqsupseteq \mathtt{e} \in \mathcal{C}$ we have $\mathtt{T} \geq \mathcal{E}(\Sigma, \mathtt{e})$, and for all $\mathtt{n}_\ell \equiv \mathtt{e} \in \mathcal{C}$ we have $\Sigma(\mathtt{n}_\ell) \equiv \mathcal{E}(\Sigma, \mathtt{e})$. Here, $\Sigma(\mathtt{e})$ is defined as follows:*

$$\begin{array}{rl} \mathcal{E}(\Sigma, \mathtt{T}) = \mathtt{T} & \\ \mathcal{E}(\Sigma, \mathtt{n}_\ell) = \mathtt{T} & \textbf{if } \{\mathtt{n}_\ell \mapsto \mathtt{T}\} \subseteq \Sigma \\ \mathcal{E}(\Sigma, \mathtt{e.f}) = \bigvee \mathtt{T}_\mathtt{i} & \textbf{if } \mathcal{E}(\Sigma, \mathtt{e}) = \bigvee\{\mathtt{T}_\mathtt{i}\ \mathtt{f}, \ldots\} \\ \mathcal{E}(\Sigma, \mathtt{e}_1[\mathtt{f} \mapsto \mathtt{e}_2]) = \bigvee\{\mathtt{T\ f}, \overline{\mathtt{T_i\ f_i}}\} & \textbf{if } \mathcal{E}(\Sigma, \mathtt{e}_1) = \bigvee\{\mathtt{T}'_\mathtt{i}\mathtt{f}, \overline{\mathtt{T_i\ f_i}}\} \textbf{ and } \mathcal{E}(\Sigma, \mathtt{e}_2) = \mathtt{T} \\ \mathcal{E}(\Sigma, \mathtt{e}_1[\mathtt{f} \mapsto \mathtt{e}_2]) = \bigvee\{\mathtt{T\ f}, \overline{\mathtt{T_i\ f_i}}\} & \textbf{if } \mathcal{E}(\Sigma, \mathtt{e}_1) = \bigvee\{\overline{\mathtt{T_i\ f_i}}\} \textbf{ and } \mathtt{f} \notin \overline{\mathtt{f}} \textbf{ and } \mathcal{E}(\Sigma, \mathtt{e}_2) = \mathtt{T} \\ \mathcal{E}(\Sigma, \neg\mathtt{e}) = \neg\mathtt{T} & \textbf{if } \overline{\mathcal{E}(\Sigma, \mathtt{e}) = \mathtt{T}} \\ \mathcal{E}(\Sigma, \bigsqcup \mathtt{e}_\mathtt{i}) = \bigvee \mathtt{T}_\mathtt{i} & \textbf{if } \overline{\mathcal{E}(\Sigma, \mathtt{e}) = \mathtt{T}} \\ \mathcal{E}(\Sigma, \bigsqcap \mathtt{e}_\mathtt{i}) = \bigwedge \mathtt{T}_\mathtt{i} & \textbf{if } \overline{\mathcal{E}(\Sigma, \mathtt{e}) = \mathtt{T}} \end{array}$$

(recall $\mathtt{T}_1 \wedge \mathtt{T}_2$ is short-hand for $\neg(\neg\mathtt{T}_1 \vee \neg\mathtt{T}_2)$)

A given FW program is considered *type safe* if a valid typing exists which satisfies all the generated typing constraints. Note, in practice, multiple satisfying typings may exist. As another example, let's see how Definition 2 supports effective record types:

```
int f({¬int f, int g} ∨ {int f, int h} x):
                                       // x0 ≡ {¬int f, int g} ∨ {int f, int h}
    x = x.f[1]                         // x1 ≡ x0.f
    return x[2]                        // int ⊒ x1
```

We have $\Sigma(\mathtt{x}_0) = \{\neg\mathtt{int\ f}, \mathtt{int\ g}\} \vee \{\mathtt{int\ f}, \mathtt{int\ h}\}$ for any valid typing, Σ, of this function. Then, it follows under Definition 2 that $\mathcal{E}(\Sigma, \mathtt{x}_0.\mathtt{f}) = \mathtt{int} \vee \neg\mathtt{int}$ (which is equivalent to `int`).

Effective unions are also supported through assignment as well. For example:

```
void f({any f,int g} ∨ {any f,int h} x):
                                        // x0 ≡ {any f, int g} ∨ {any f, int h}
        y = 1                           // y0 ≡ int
        x.f = y1                        // x1 ≡ x0[f ↦ y0]
```

Here, $\mathcal{E}(\Sigma, \mathtt{x}_0[\mathtt{f} \mapsto \mathtt{y}_0]) = \{\mathtt{int\ f}, \mathtt{int\ g}\} \vee \{\mathtt{int\ f}, \mathtt{int\ h}\}$ for any valid typing, Σ, of this function. Thus, the above example type checks.

4.1 Typing Rules

Figure 3 gives the constraint-based typing rules for FW which have a general form of $\Gamma_0 \vdash \mathtt{S} : \Gamma_1 \lfloor \mathcal{C}$ (except T-FUN, which is similar). Here, Γ_0 represents the typing environment immediately before S, whilst Γ_1 represents that immediately after. In the constraint-based formulation, a typing environment Γ maps each variable to the program point where its current value was defined. Finally, $\mathcal{C}$ is the constraint set which must hold (i.e. admit a valid solution) for that statement to be type safe.

T-FUN initialises the typing environment from the parameter types, and employs a special variable, \$, to connect the return type with any returned values (via T-RV). This is done by mapping \$ to the return type T in Γ, so it can be recalled in T-RV. The following illustrates:

```
int f(any x):
                                        // x0 ≡ any  (T-FUN)
    x = 11                              // x1 ≡ int  (T-VC)
    return x2                           // int ⊒ x1  (T-RV)
```

Here, $\mathtt{x}_1$ is connected to the return type through \$. Rule T-VC constrains the type of the assigned variable to that of the assigned (constant) value. The environment produced (i.e. $\Gamma[\mathtt{n} \mapsto \ell]$) equals the old (i.e. Γ) but with n mapped to ℓ. Rule T-VV constrains the type of the assigned variable to that of the right-hand side. Here, $\Gamma(\mathtt{m}) = \kappa$ determines the program point (κ) where the type variable currently representing m was defined ($\mathtt{m}_\kappa$).

Rule T-VF is similar to T-VC, but instead constrains the assigned variable to the corresponding field of the right-hand side. Rule T-FV uses a constraint of the form $\mathtt{n}_\ell \equiv \mathtt{n}_\kappa[\mathtt{f} \mapsto \mathtt{m}_\lambda]$. This constrains all fields of $\mathtt{n}_\ell$ (except for f) to their corresponding type in $\mathtt{n}_\kappa$, whilst field f now maps to $\mathtt{m}_\lambda$.

Rule T-IF is the fairly involved. This employs a support function, defs(B), to identify variables assigned in B. The variable being tested, n, is split into type variables $\mathtt{n}_{\ell+}$ and $\mathtt{n}_{\ell-}$ for the true branch and false branch (respectively). Furthermore, for any variable m defined within $\mathtt{B}_1$ or $\mathtt{B}_2$, a constraint $\mathtt{m}_\ell \equiv \mathtt{m}_\kappa \sqcup \mathtt{m}_\lambda$ is added to join the flow from both branches, where $\mathtt{m}_\kappa$ gives the flow of m from $\mathtt{B}_1$ and $\mathtt{m}_\lambda$ gives the flow of m from $\mathtt{B}_2$.

Finally, rule T-WHILE is similar, in some ways, to T-IF. Each variable $\mathtt{n} \in \mathtt{defs(B)}$ requires a constraint to merge flow from *before* the loop (i.e. $\mathtt{n}_\kappa$) with that from *around* the loop (i.e. $\mathtt{n}_\lambda$). Here, $\mathtt{n}_\ell$ is created to capture this flow and, hence, n maps to ℓ in the resulting environment (i.e. Γ^1). This corresponds to the placement of $\phi-$nodes in SSA form [65–67]. However, our setting is simpler as we do not have unstructured control-flow.

4.2 Constraint Solving

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

The essence of our approach is, for each variable $\mathtt{n}_\ell$, to generate a *single constraint* determining its solution. This requires us to have exactly one constraint of the form $\mathtt{n}_\ell \equiv \mathtt{e}$ for each variable $\mathtt{n}_\ell$:

Definition 3 (Variable Scoping) *Let $\mathcal{C}_\mathcal{X}$ denote a constraint set where $\mathcal{X}$ defines the variables permissible in any constraint $\mathtt{T} \sqsupseteq \mathtt{e} \in \mathcal{C}_\mathcal{X}$ or $\mathtt{n}_\ell \equiv \mathtt{e} \in \mathcal{C}_\mathcal{X}$.*

Definition 4 (Single Assignment) *A constraint set $\mathcal{C}_\mathcal{X}$ is in* single assignment form *if, for each $\mathtt{n}_\ell \in \mathcal{X}$, there is exactly one constraint in $\mathcal{C}_\mathcal{X}$ of the form $\mathtt{n}_\ell \equiv \mathtt{e}$.*

Function Typing (constraints):

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

Block Typing (constraints):

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

Statement Typing (constraints):

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

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

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

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

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

$$\frac{\begin{array}{c}\Gamma^0[\mathtt{n} \mapsto \ell^+] \vdash \mathtt{B}_1 : \Gamma^1 \mid \mathcal{C}_1 \\ \Gamma^0[\mathtt{n} \mapsto \ell^-] \vdash \mathtt{B}_2 : \Gamma^2 \mid \mathcal{C}_2 \\ \overline{\mathtt{m}} = \Gamma^1 \otimes \Gamma^2 \quad \mathcal{C}_3 = \mathrm{join}(\ell, \Gamma^1, \Gamma^2) \\ \mathcal{C}_4 = \{\mathtt{n}_{\ell+} \equiv \mathtt{n}_\kappa \sqcap \mathtt{T}, \mathtt{n}_{\ell-} \equiv \mathtt{n}_\kappa \sqcap \neg\mathtt{T}\}\end{array}}{\Gamma^0 \vdash \mathtt{if}\ [\![\mathtt{n\ is\ T}]\!]^{\ell} : \mathtt{B}_1\ \mathtt{else} : \mathtt{B}_2 : \Gamma^0[\overline{\mathtt{m} \mapsto \ell}] \mid \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3 \cup \mathcal{C}_4} \quad \text{(T-IF)}$$

$$\frac{\begin{array}{c}\Gamma^0 \vdash \mathtt{B} : \Gamma^1 \mid \mathcal{C}_1 \quad \overline{\mathtt{m}} = \Gamma^0 \otimes \Gamma^1 \\ \Gamma^2 = \Gamma^0[\overline{\mathtt{m} \mapsto \ell}] \quad \Gamma^2 \vdash \mathtt{B} : \Gamma^3 \mid \mathcal{C}_2 \\ \mathcal{C}_3 = \mathtt{join}(\ell, \Gamma^0, \Gamma^1) \\ \Gamma^1(\mathtt{n}) = \kappa \quad \Gamma^1(\mathtt{m}) = \lambda \quad \mathcal{C}_4 = \{\mathtt{n}_\kappa \equiv \mathtt{int}, \mathtt{m}_\lambda \equiv \mathtt{int}\}\end{array}}{\Gamma^0 \vdash \mathtt{while}\ [\![\mathtt{n} < \mathtt{m}]\!]^{\ell} : \mathtt{B} : \Gamma^0 \mid \mathcal{C}_2 \cup \mathcal{C}_3 \cup \mathcal{C}_4} \quad \text{(T-WHILE)}$$

Helpers:

$$\Gamma^1 \otimes \Gamma^2 = \{\mathtt{m} \mid \mathtt{m} \mapsto \kappa \in \Gamma^1 \wedge \mathtt{m} \mapsto \lambda \in \Gamma^2 \wedge \kappa \neq \lambda\}$$
$$\mathrm{join}(\ell, \Gamma^1, \Gamma^2) = \{\mathtt{m}_\ell \equiv \mathtt{m}_\kappa \sqcup \mathtt{m}_\lambda \mid \mathtt{m} \in \Gamma^1 \otimes \Gamma^2 \wedge \kappa = \Gamma^1(\mathtt{m}) \wedge \lambda = \Gamma^2(\mathtt{m})\}$$

Figure 3: Constraint-Based Typing rules for FW

We now observe that any constraint set $\mathcal{C}_\mathcal{X}$ generated from the rules of Figure 3 is guaranteed to be in single assignment form. This is because each constraint of the form $\mathtt{n}_\ell \equiv \mathtt{e}$ is tied to a unique program point ℓ. Furthermore, none of the rules from Figure 3 can generate multiple constraints for the same variable.

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

Definition 5 (Elimination Step) *Let $\mathcal{C}_\mathcal{X}$ be a constraint set in single assignment form, where $\mathtt{n}_\ell \equiv \mathtt{e}_1 \in \mathcal{C}_\mathcal{X}$. Then, $\mathcal{C}_{\mathcal{X}-\{\mathtt{n}_\ell\}} = \{\mathtt{n}_\kappa \equiv \mathtt{e}_2[\![\mathtt{n}_\ell \mapsto \mathtt{e}_1]\!] \mid \mathtt{n}_\kappa \equiv \mathtt{e}_2 \in \mathcal{C}_\mathcal{X} \wedge \ell \neq \kappa\} \cup \{\mathtt{T} \sqsupseteq \mathtt{e}_2[\![\mathtt{n}_\ell \mapsto \mathtt{e}_1]\!] \mid \mathtt{T} \sqsupseteq \mathtt{e}_2 \in \mathcal{C}_\mathcal{X}\}$.*

Here, the choice of $\mathtt{n}_\ell$ to eliminate is arbitrary. Furthermore, $\mathtt{e}_2[\![\mathtt{n}_\ell \mapsto \mathtt{e}_1]\!]$ substitutes all occurrences of $\mathtt{n}_\ell$ with $\mathtt{e}_1$ in $\mathtt{e}_2$. To determine the typing for a given variable $\mathtt{n}_\ell$, we progressively eliminate variables until only $\mathtt{n}_\ell$ remains. Then, we have $\mathtt{n}_\ell \equiv \mathtt{e} \in \mathcal{C}_{\{\mathtt{n}_\ell\}}$ and $\mathcal{E}(\emptyset, \mathtt{e})$ (if it is well-defined) gives the typing for $\mathtt{n}_\ell$:

Definition 6 (Elimination Typing) *Let $\mathcal{C}_\mathcal{X} \vdash \mathtt{n}_\ell : \mathtt{T}_\ell$ denote for $\mathtt{n}_\ell \in \mathcal{X}$ that some sequence of zero of more eliminations exists which reduce $\mathcal{C}_\mathcal{X}$ to $\mathcal{C}_{\{\mathtt{n}_\ell\}}$ where $\mathtt{n}_\ell \equiv \mathtt{e} \in \mathcal{C}_{\{\mathtt{n}_\ell\}}$ and $\mathtt{T}_\ell = \mathcal{E}(\emptyset, \mathtt{e})$.*

The variable elimination process is trivially guaranteed to terminate. However, it will fail in the case of an untypeable program. For example, $\mathcal{E}(\emptyset, \mathtt{int.f})$ is not well defined. Likewise, examples which produce *recursive* constraints are untypeable, such as:

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

This example generates the recursive constraint $\mathtt{z}_3 \equiv \mathtt{z}_3[\mathtt{f} \mapsto \mathtt{z}_3] \sqcup \{\mathtt{int\ f}\}$. Such constraints have no solution (i.e. $\mathcal{E}(\emptyset, \mathtt{z}_3[\mathtt{f} \mapsto \mathtt{z}_3])$ is not well defined). This example also serves to illustrate why our flow-typing system is presented using *constraint-based* rules, rather than the more traditional *dataflow-based* rules. The reason is that the dataflow-based typing rules will not terminate on the above example, since there is no fixed-point solution. We discuss this in more detail in our earlier work [38, 39].

Finally, an important property to show for our variable elimination process is that it is *complete*. That is, any valid typing can be found via elimination.

Lemma 1 (Substitution Equivalence) *Let Σ be a typing and $\mathcal{C}_\mathcal{X}$ be a constraint set in single assignment form, where $\Sigma \models \mathcal{C}_\mathcal{X}$ and $\mathtt{n}_\ell \equiv \mathtt{e}_1 \in \mathcal{C}_\mathcal{X}$. Then, for any $\mathtt{e}_2$ where $\mathcal{E}(\Sigma, \mathtt{e}_2)$ is well-defined, it follows that $\mathcal{E}(\Sigma, \mathtt{e}_2) \equiv \mathcal{E}(\Sigma, \mathtt{e}_2[\![\mathtt{n}_\ell \mapsto \mathtt{e}_1]\!])$].*

Proof 1 *By structural induction on $\mathtt{e}_2$, where the induction hypothesis states that the Lemma holds for any substructure of $\mathtt{e}_2$:*

- *Case $\mathtt{e}_2 = \mathtt{T}$: Straightforward since $\mathtt{e}_2[\![\mathtt{n}_\ell \mapsto \mathtt{e}_1]\!] = \mathtt{e}_2$.*
- *Case $\mathtt{e}_2 = \mathtt{m}_\kappa$: if $\mathtt{m}_\kappa \neq \mathtt{n}_\ell$ then $\mathtt{e}_2[\![\mathtt{n}_\ell \mapsto \mathtt{e}_1]\!] = \mathtt{e}_2$. Otherwise, $\mathtt{e}_2 = \mathtt{n}_\ell$ and $\mathtt{e}_2[\![\mathtt{n}_\ell \mapsto \mathtt{e}_1]\!] = \mathtt{e}_1$. By assumption $\Sigma \models \mathcal{C}_\mathcal{X}$ and $\mathtt{n}_\ell \equiv \mathtt{e}_1 \in \mathcal{C}_\mathcal{X}$, which implies $\Sigma(\mathtt{n}_\ell) \equiv \mathcal{E}(\Sigma, \mathtt{e}_1)$ under Definition 2.*
- *Case $\mathtt{e}_2 = \mathtt{e}_3.\mathtt{f}$: By induction, $\mathcal{E}(\Sigma, \mathtt{e}_3) \equiv \mathcal{E}(\Sigma, \mathtt{e}_3[\![\mathtt{n}_\ell \mapsto \mathtt{e}_1]\!])$. Thus, $\mathcal{E}(\Sigma, \mathtt{e}_3.\mathtt{f}) \equiv \mathcal{E}(\Sigma, \mathtt{e}_3[\![\mathtt{n}_\ell \mapsto \mathtt{e}_1]\!].\mathtt{f})$ follows immediately.*
- *Case $\mathtt{e}_2 = \mathtt{e}_3[\mathtt{f} \mapsto \mathtt{e}_4]$: By induction, $\mathcal{E}(\Sigma, \mathtt{e}_3) \equiv \mathcal{E}(\Sigma, \mathtt{e}_3[\![\mathtt{n}_\ell \mapsto \mathtt{e}_1]\!])$ and $\mathcal{E}(\Sigma, \mathtt{e}_4) \equiv \mathcal{E}(\Sigma, \mathtt{e}_4[\![\mathtt{n}_\ell \mapsto \mathtt{e}_1]\!])$. Thus, $\mathcal{E}(\Sigma, \mathtt{e}_3[\mathtt{f} \mapsto \mathtt{e}_4]) \equiv \mathcal{E}(\Sigma, \mathtt{e}_3[\![\mathtt{n}_\ell \mapsto \mathtt{e}_1]\!][\mathtt{f} \mapsto \mathtt{e}_4[\![\mathtt{n}_\ell \mapsto \mathtt{e}_1]\!]])$ follows immediately.*
- *Case $\mathtt{e}_2 = \neg\mathtt{e}_3$: By induction, $\mathcal{E}(\Sigma, \mathtt{e}_3) \equiv \mathcal{E}(\Sigma, \mathtt{e}_3[\![\mathtt{n}_\ell \mapsto \mathtt{e}_1]\!])$. Thus, $\mathcal{E}(\Sigma, \neg\mathtt{e}_3) \equiv \mathcal{E}(\Sigma, (\neg\mathtt{e}_3)[\![\mathtt{n}_\ell \mapsto \mathtt{e}_1]\!])$ follows immediately.*
- *Case $\mathtt{e}_2 = \bigsqcup \mathtt{e}_\mathtt{i}$: By induction, $\forall \mathtt{i}.\mathcal{E}(\Sigma, \mathtt{e}_\mathtt{i}) \equiv \mathcal{E}(\Sigma, \mathtt{e}_\mathtt{i}[\![\mathtt{n}_\ell \mapsto \mathtt{e}_1]\!])$. Thus, $\mathcal{E}(\Sigma, \bigsqcup \mathtt{e}_\mathtt{i}) \equiv \mathcal{E}(\Sigma, \bigsqcup \mathtt{e}_\mathtt{i}[\![\mathtt{n}_\ell \mapsto \mathtt{e}_1]\!])$ follows immediately.*
- *Case $\mathtt{e}_2 = \bigsqcap \mathtt{e}_\mathtt{i}$: By induction, $\forall \mathtt{i}.\mathcal{E}(\Sigma, \mathtt{e}_\mathtt{i}) \equiv \mathcal{E}(\Sigma, \mathtt{e}_\mathtt{i}[\![\mathtt{n}_\ell \mapsto \mathtt{e}_1]\!])$. Thus, $\mathcal{E}(\Sigma, \bigsqcap \mathtt{e}_\mathtt{i}) \equiv \mathcal{E}(\Sigma, \bigsqcap \mathtt{e}_\mathtt{i}[\![\mathtt{n}_\ell \mapsto \mathtt{e}_1]\!])$ follows immediately.*

□

Theorem 1 (Elimination Completeness) *Let $\mathcal{C}_\mathcal{X}$ be a constraint set in single assignment form. If $\Sigma \models \mathcal{C}_\mathcal{X}$ then, for any $\mathtt{n}_\ell \in \mathcal{X}$, it follows that $\mathcal{C}_\mathcal{X} \vdash \mathtt{n}_\ell : \mathtt{T}_\ell$ where $\Sigma(\mathtt{n}_\ell) \equiv \mathtt{T}_\ell$.*

Proof 2 *By induction on the size of $\mathcal{X}$. The induction hypothesis states the Theorem holds for any $\mathcal{C}_\mathcal{X}$ where $|\mathcal{X}| \leq \mathtt{k}$:*

- *Case $\mathtt{k} = 0$. Follows immediately since $\mathcal{X} = \emptyset$.*
- *Case $\mathtt{k} > 0$. Let $\mathtt{n}_\ell \equiv \mathtt{e}_1 \in \mathcal{C}_\mathcal{X}$. Consider an arbitrary $\mathtt{c} \in \mathcal{C}_\mathcal{X}$ of the form $\mathtt{T} \sqsupseteq \mathtt{e}_2$ or $\mathtt{m}_\kappa \equiv \mathtt{e}_2$ (where $\mathtt{m}_\kappa \neq \mathtt{n}_\ell$). By assumption $\Sigma \models \{\mathtt{c}\}$ and $\Sigma(\mathtt{n}_\ell) \equiv \mathcal{E}(\Sigma, \mathtt{e}_1)$. By Lemma 1, $\mathcal{E}(\Sigma, \mathtt{e}_2) \equiv \mathcal{E}(\Sigma, \mathtt{e}_2[\![\mathtt{n}_\ell \mapsto \mathtt{e}_1]\!])$ and, hence, $\Sigma \models \{\mathtt{c}[\![\mathtt{n}_\ell \mapsto \mathtt{e}_1]\!]\}$. Therefore, $\Sigma \models \mathcal{C}_{\mathcal{X}-\{\mathtt{n}_\ell\}}$.*

□

A common concept which is analogous to completeness is that of *soundness*. Hence, one may wonder whether a corresponding soundness theorem for variable elimination is required. In fact, this is not necessary because a typing, Σ, obtained via variable elimination acts as a *certificate*. That is, following Definition 2, we can easily test whether $\Sigma \models \mathcal{C}_\mathcal{X}$ for the original constraint set $\mathcal{C}_\mathcal{X}$. Furthermore, it is easy to see that variable elimination may indeed producing a typing Σ where $\Sigma \not\models \mathcal{C}_\mathcal{X}$. For example, if $\mathcal{C}_\mathcal{X} = \{\mathtt{n}_\ell \equiv \mathtt{int}, \mathtt{int} \sqsupseteq \mathtt{any}\}$ then $\mathcal{C}_\mathcal{X} \vdash \mathtt{n}_\ell : \mathtt{int}$, but clearly no Σ exists where $\Sigma \models \mathcal{C}_\mathcal{X}$.

4.3 Progress and Preservation

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

Definition 7 (Safe Abstraction) *Let (Σ, Γ) be a typing and environment and Δ a runtime environment. Then, (Σ, Γ) safely abstracts Δ, denoted $(\Sigma, \Gamma) \approx \Delta$, iff $\mathbf{dom}(\Gamma) \subseteq \mathbf{dom}(\Delta)$ and, for all $\mathtt{n} \mapsto \ell \in \Gamma$, it holds that $\Sigma(\mathtt{n}_\ell) \models \Delta(\mathtt{n})$.*

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

Theorem 2 (Progress) *Assume Δ, Σ and Γ where $(\Sigma, \Gamma) \approx \Delta$ holds. If $\Gamma \vdash \mathtt{S} : \Gamma' \downharpoonright \mathcal{C}$ and $\Sigma \models \mathcal{C}$, then either $\langle \Delta, \mathtt{S}\ \mathtt{B} \rangle \longrightarrow \langle \Delta', \mathtt{B}' \rangle$ or $\langle \Delta, \mathtt{S}\ \mathtt{B} \rangle \longrightarrow \mathtt{halt}(\mathtt{v})$.*

Proof 3 *By case analysis on `S` over the different statement forms from Figure 1.*

- *Case "$[\![\mathtt{n} = \mathtt{v}]\!]^\ell$ `B`": Straightforward, since rule R-VC has no antecedents.*
- *Case "$[\![\mathtt{n} = \mathtt{m}]\!]^\ell$ `B`": R-VV requires $\Delta(\mathtt{m})$ be well-defined. This follows from rule T-VV which requires $\Gamma(\mathtt{m})$ be well-defined.*
- *Case "$[\![\mathtt{n} = \mathtt{m.f}]\!]^\ell$ `B`": R-VF requires $\Delta(\mathtt{m.f}) = \{\mathtt{f} : \mathtt{v}, \ldots\}$. This follows from T-VF, as $\Sigma \models \{\mathtt{n}_\ell \equiv \mathtt{m}_\kappa.\mathtt{f}\}$ requires $\mathcal{E}(\Sigma, \mathtt{m}_\kappa.\mathtt{f})$ be well-defined. This implies $\Sigma(\mathtt{m}_\kappa) = \bigvee\{\mathtt{T\ f}, \ldots\}$ by Definition 2.*
- *Case "$[\![\mathtt{n.f} = \mathtt{m}]\!]^\ell$ `B`": R-FV requires $\Delta(\mathtt{m})$ be well-defined and $\Delta(\mathtt{n}) = \{\overline{\mathtt{f} : \mathtt{v}}\}$. The former follows as for R-VV. The latter from T-FV, as $\Sigma \models \{\mathtt{n}_\ell \equiv \mathtt{n}_\kappa[\mathtt{f} \mapsto \mathtt{m}_\lambda]\}$ requires $\mathcal{E}(\Sigma, \mathtt{n}_\kappa[\mathtt{f} \mapsto \mathtt{m}_\lambda])$ be well-defined. This implies $\Sigma(\mathtt{n}_\kappa) = \bigvee\{\overline{\mathtt{T\ f}}\}$ by Definition 2.*
- *Case "$[\![\mathtt{return\ n}]\!]^\ell$ `B`": R-RV requires $\Delta(\mathtt{n})$ be well-defined. This follows from rule T-RV which requires $\Gamma(\mathtt{n})$ be well-defined.*
- *Case "$\mathtt{if}\ [\![\mathtt{n\ is\ T}]\!]^\ell\ \mathtt{:\ B_1\ else\ B_2\ ;\ B_3}$": R-I1 and R-I2 require only that $\Delta(\mathtt{n})$ be well-defined. This follows from rule T-IF which requires $\Gamma(\mathtt{n})$ be well-defined.*

- *Case* "$\texttt{while}\,[\![\texttt{n} < \texttt{m}]\!]^{\ell}\colon \texttt{B}_1\ ;\ \texttt{B}_2$"*: R-W1 and R-W2 require* $\Gamma(\texttt{n})$ *and* $\Gamma(\texttt{m})$ *yield* `int` *values. This follows from T-WHILE as* $\Sigma \models \{\texttt{int} \equiv \texttt{n}_\kappa, \texttt{int} \equiv \texttt{m}_\lambda\}$ *implies* $\texttt{int} \equiv \Sigma(\texttt{n}_\kappa)$ *and* $\texttt{int} \equiv \Sigma(\texttt{m}_\lambda)$.

□

Theorem 3 (Preservation) *Assume* Δ, Σ *and* Γ *where* $(\Sigma, \Gamma) \approx \Delta$ *holds. If* $\Gamma \vdash \texttt{S} : \Gamma' \lfloor \mathcal{C}$, $\Sigma \models \mathcal{C}$ *and* $\langle \Delta, \texttt{S}\ \texttt{B}\rangle \longrightarrow \langle \Delta', \texttt{B}'\rangle$, *then* $(\Sigma, \Gamma') \approx \Delta'$.

Proof 4 *By case analysis on* S *over the different statement forms from Figure 1.*

- *Case* "$[\![\texttt{n} = \texttt{v}]\!]^{\ell}\ \texttt{B}$" *where* $\vdash \texttt{v} : \texttt{T}$, $\Delta' = \Delta[\texttt{n} \mapsto \texttt{T}]$ *and* $\Gamma' = \Gamma[\texttt{n} \mapsto \ell]$*: This follows from T-VC as* $\Sigma \models \{\texttt{n}_\ell \equiv \texttt{T}\}$ *implies* $\Sigma(\texttt{n}_\ell) \equiv \texttt{T}$ *under Definition 2. Therefore,* $(\Sigma, \Gamma[\texttt{n} \mapsto \ell]) \approx \Delta[\texttt{n} \mapsto \texttt{T}]$.

- *Case* "$[\![\texttt{n} = \texttt{m}]\!]^{\ell}\ \texttt{B}$" *where* $\Gamma(\texttt{m}) = \kappa$, $\Delta' = \Delta[\texttt{n} \mapsto \Delta(\texttt{m})]$ *and* $\Gamma' = \Gamma[\texttt{n} \mapsto \ell]$*: This follows from T-VV as* $\Sigma \models \{\texttt{n}_\ell \equiv \texttt{m}_\kappa\}$ *implies* $\Sigma(\texttt{n}_\ell) \equiv \Sigma(\texttt{m}_\kappa)$ *under Definition 2. Therefore,* $(\Sigma, \Gamma[\texttt{n} \mapsto \ell]) \approx \Delta[\texttt{n} \mapsto \Delta(\texttt{m})]$.

- *Case* "$[\![\texttt{n} = \texttt{m.f}]\!]^{\ell}\ \texttt{B}$" *where* $\Gamma(\texttt{m}) = \kappa$, $\Delta' = \Delta[\texttt{n} \mapsto \Delta(\texttt{m}).\texttt{f}]$ *and* $\Gamma' = \Gamma[\texttt{n} \mapsto \ell]$*: This follows from T-VF as* $\Sigma \models \{\texttt{n}_\ell \equiv \texttt{m}_\kappa.\texttt{f}\}$ *implies* $\Sigma(\texttt{n}_\ell) \equiv \Sigma(\texttt{m}_\kappa.\texttt{f})$ *under Definition 2. Thus,* $(\Sigma, \Gamma[\texttt{n} \mapsto \ell]) \approx \Delta[\texttt{n} \mapsto \Delta(\texttt{m}).\texttt{f}]$.

- *Case* "$[\![\texttt{n.f} = \texttt{m}]\!]^{\ell}\ \texttt{B}$" *where* $\Gamma(\texttt{n}) = \kappa$, $\Gamma(\texttt{m}) = \lambda$, $\texttt{v} = \Delta(\texttt{n})[\texttt{f} \mapsto \Delta(\texttt{m})]$, $\Delta' = \Delta[\texttt{n} \mapsto \texttt{v}]$ *and* $\Gamma' = \Gamma[\texttt{n} \mapsto \ell]$*: This follows from T-FV as* $\Sigma \models \{\texttt{n}_\ell \equiv \texttt{n}_\kappa[\texttt{f} \mapsto \texttt{m}_\lambda]\}$ *implies* $\Sigma(\texttt{n}_\ell) \equiv \Sigma(\texttt{n}_\kappa[\texttt{f} \mapsto \texttt{m}_\lambda])$ *under Definition 2. Furthermore,* $\Sigma(\texttt{n}_\kappa) \models \Delta(\texttt{n})$ *and* $\Sigma(\texttt{m}_\lambda) \models \Delta(\texttt{m})$ *by construction and, thus,* $(\Sigma, \Gamma[\texttt{n} \mapsto \ell]) \approx \Delta[\texttt{n} \mapsto \texttt{v}]$.

- *Case* "$[\![\texttt{return n}]\!]^{\ell}\ \texttt{B}$"*: This follows immediately from R-RV because it does not produce a successor state* $\langle \Delta', \texttt{B}'\rangle$.

- *Case* "$\texttt{if}\,[\![\texttt{n is T}]\!]^{\ell}\colon \texttt{B}_1\ \texttt{else}\ \texttt{B}_2\ ;\ \texttt{B}_3$" *where* $\texttt{B}' = \texttt{B}_1$, $\Delta' = \Delta$ *and* $\Gamma' = \Gamma[\texttt{n} \mapsto \ell^+]$*: This follows from T-IF as* $\Sigma \models \{\texttt{n}_{\ell+} \equiv \texttt{n}_\kappa \sqcap \texttt{T}\}$ *implies* $\Sigma(\texttt{n}_{\ell+}) \equiv \Sigma(\texttt{n}_\kappa) \wedge \texttt{T}$ *giving* $\Sigma(\texttt{n}_{\ell+}) \leq \Sigma(\texttt{n}_\kappa)$ *and* $\Sigma(\texttt{n}_{\ell+}) \leq \texttt{T}$ *by S-UNION2 and S-NEG1 (recall* $\texttt{T}_1 \wedge \texttt{T}_2$ *is sugar for* $\neg(\neg\texttt{T}_1 \vee \neg\texttt{T}_2)$*). Hence,* $(\Sigma, \Gamma[\texttt{n} \mapsto \ell^+]) \approx \Delta$.

- *Case* "$\texttt{if}\,[\![\texttt{n is T}]\!]^{\ell}\colon \texttt{B}_1\ \texttt{else}\ \texttt{B}_2\ ;\ \texttt{B}_3$" *where* $\texttt{B}' = \texttt{B}_2$, $\Delta' = \Delta$ *and* $\Gamma' = \Gamma[\texttt{n} \mapsto \ell^-]$*: This follows from T-IF as* $\Sigma \models \{\texttt{n}_{\ell-} \equiv \texttt{n}_\kappa \sqcap \neg\texttt{T}\}$ *implies* $\Sigma(\texttt{n}_{\ell-}) \equiv \Sigma(\texttt{n}_\kappa) \wedge \neg\texttt{T}$ *which gives* $\Sigma(\texttt{n}_{\ell-}) \leq \Sigma(\texttt{n}_\kappa)$ *and* $\Sigma(\texttt{n}_{\ell-}) \leq \neg\texttt{T}$ *by S-UNION2 and S-NEG1 (recalling* $\texttt{T}_1 \wedge \texttt{T}_2$ *is sugar for* $\neg(\neg\texttt{T}_1 \vee \neg\texttt{T}_2)$*). Hence,* $(\Sigma, \Gamma[\texttt{n} \mapsto \ell^-]) \approx \Delta$.

- *Case* "$\texttt{while}\,[\![\texttt{n} < \texttt{m}]\!]^{\ell}\{\texttt{B}_1\}\ \texttt{B}_2$" *where* $\Delta' = \Delta$ *and* $\Gamma' = \Gamma[\overline{\texttt{m} \mapsto \ell}]$*: This follows from T-WHILE as* $\Sigma \models \{\overline{\texttt{m}_\ell \equiv \texttt{m}_\kappa \sqcup \texttt{m}_\lambda}\}$ *implies* $\overline{\Sigma(\texttt{m}_\ell) \equiv \Sigma(\texttt{m}_\kappa) \vee \Sigma(\texttt{m}_\lambda)}$ *which, by S-UNION2, gives* $\overline{\Sigma(\texttt{m}_\ell) \geq \Sigma(\texttt{m}_\kappa)}$ *and* $\overline{\Sigma(\texttt{m}_\ell) \geq \Sigma(\texttt{m}_\lambda)}$*. Thus,* $\overline{\Sigma(\texttt{m}_\ell) \geq \Delta(\texttt{n})}$ *regardless of the whether the previous statement was from* before *the loop or from* around *the loop and, hence,* $(\Sigma, \Gamma[\overline{\texttt{n} \mapsto \ell}]) \approx \Delta$.

□

ISSUES. There some known issues with these proofs. The proof of Theorem 3 does not account for the merging of control-flow at the end of an **`if`** statements (perhaps need a big-step operational semantics?) Also, need a stronger notion between $\Gamma^1 \otimes \Gamma^2$ and the runtime notion of a variable being defined (perhaps as a Lemma?).

5 Implementation and Results

In this section, we report on experimental results examining the cost of employing value semantics. The Whiley compiler targets the JVM and can be downloaded, along with the benchmarks discussed here, from `http://whiley.org`.

5.1 Implementation

Our compiler faces a number of challenges in compiling Whiley to the JVM. In particular, flow- and structural-typing goes against the grain of the JVM. For example, explicit casts must be inserted after type tests, and care must be taken to properly coerce data types at control-flow merges. These issues are beyond the scope of this paper, and more details can be found in [30].

Here, we focus on optimising updateable value semantics — this has not been discussed in [30] or elsewhere. We use JVM objects to represent Whiley's compound types (lists, sets, maps, and records) and deferred reference counting [68] to eliminate unnecessary copies. The implementation of reference counting is straightforward: objects are created with a reference count of one; reference counts are incremented when objects are used in expressions and decremented when variables referring to them are no longer live; an object's reference count is incremented when it is read out from an enclosing object; and reference counts of all contained objects are decremented when the reference count of their enclosing object reaches zero. An intra-procedural live variables analysis is used to determine when a variable is and is not live.

The key optimisation applies when a compound object is updated, that is, when an element/field is written to a set, list, map, or record. If the enclosing object's reference count is exactly one, then the update is performed in place. If the enclosing object's reference count is two or more, the object is cloned and the clone updated. The following illustrates:

```
1  [int] f([int] z):
2      z[0] = 1 // Object #2 created
3      return z
4
5  [int] g():
6      x = [1,2,3] // Object #1 created
7      y = f(x)
8      return x + y
```

On Line 6, Object #1 which represents the constructed list has an initial reference count of one. This does not change when it is assigned to `x`. Its reference count is then increased by one on Line 7, as `x` is used in the invocation expression and remains live afterwards. On entry to `f()`, parameter `z` refers to Object #1, which now has reference count two. Therefore, the list assignment on Line 2 creates an entirely new object before updating it. It also decrements the reference count of Object #1. On Line 8, `x` still refers to Object #1 (now with reference count one) and, hence, the append is performed in place on it.

Finally, our implementation makes no effort to reduce the cost of checking reference counts (e.g. as in [31]). For example:

```
while i < |list|:
    list[i] = null
    i = i + 1
```

The reference count of the object referred to by `list` will be checked on every iteration of this loop. In fact, it need only be checked once and, hence, could be hoisted out. This is because, after the first check, the reference count is guaranteed to be one.

5.2 Experiment

The aim of our experiment was to determine how often our implementation performs in place updates of compound structures. The following update operations were considered:

- **Lists.** The operations considered here were *assignment* (e.g. `l[i]=e`), *append* (e.g. `l1+l2`) and *sublist* (e.g. `l[1..10]`). The benefit of performing an in place list assignment is that we avoid cloning the list being assigned. The benefit from an in place append stems from reusing one of the

Name	Description	LOC	Clones (%)	Refs	Size	Time (s)
Jasm	A Java bytecode disassembler. The benchmark reads a Java class file (91KB) and disassembles it (much like `javap -verbose`).	2333	43.0% $\left(\frac{12878}{29968}\right)$	8.8	8.0	2.1s (2.5s)
Gunzip	An implementation of the DEFLATE decompression algorithm. The benchmark reads a gzipped file (11KB) and decompresses it.	815	0.62% $\left(\frac{873}{140561}\right)$	1.01	2.3	1.6s (39.8s)
Chess	Validates chess games in short algebraic notation adhere the rules of chess. The benchmark runs over the longest chess game in history, Nikoli-Arsovi.	784	1.6% $\left(\frac{6438}{416116}\right)$	1.1	7.4	1.4s (1.5s)
Calc	An arithmetic expression evaluator. This reads in a file of 1000 randomly generated expressions and evaluates them.	225	0.0% $\left(\frac{0}{81527}\right)$	1.0	-	8.8s (8.9s)
SCC	Tarjan's algorithm for finding strong components. The benchmark reads 10 randomly generated digraphs of 1000 nodes and finds their components.	169	4.86% $\left(\frac{12602}{258968}\right)$	26.2	315	35s (38.6s)
Matrix	A simple matrix multiplication algorithm. The benchmark reads in two randomly generated matrices of size 100x100 and multiplies them.	94	0.0% $\left(\frac{0}{30302}\right)$	1.0	-	6.0s (6.0s)
CJ-A	A solution to Google Code Jam problem A from May 2011. The benchmark reads in the large data set (35KB) and solves it.	87	0.0% $\left(\frac{0}{7158}\right)$	1.0	-	2.2s (2.2s)
LZ77	A simple implementation of the well-known Lempel-Ziv algorithm. This benchmark reads a file (74KB), compresses it and then decompresses it.	82	0.0% $\left(\frac{0}{120500}\right)$	1.0	-	9.4s (226s)
Sort	A simple implementation of merge-sort. The benchmark reads in a file with 10000 numbers randomly arranged and sorts them.	64	12.2% $\left(\frac{19998}{163614}\right)$	1.12	6.7	5.2s (18.4s)
Queens	A simple algorithm for solving the well-known N-Queens problem. The benchmark solves for boards of size 10x10.	44	36.3% $\left(\frac{25814}{71086}\right)$	1.24	10.0	0.7s (1.4s)

Table 1: The benchmark suite and experimental results. Here, "LOC" counts the Lines of Code, "Clones" reports the ratio of update operations which required a clone, "Refs" reports the average reference count of an operand to an update operation, "Size" reports the average size of a cloned object, and "Time" gives the benchmark's execution time with (and without) reference counting.

two operands, rather than creating a fresh list from scratch. Likewise, the benefit from performing an in place sublist operation is that we avoid cloning the list and can, instead, simply discard elements.

- **Records and Maps.** The operation considered here was *assignment*. As for lists, the benefit of performing an in place assignment is that we avoid cloning the structure being updated.

- **Sets.** The operations considered here were *union*, *intersection* and *difference*. As for list append, we can benefit here by reusing one of the two operands, rather than creating a new set from scratch.

For each of our benchmarks we recorded the total number of the above operations, along with the number performed in place. Fewer in place updates means more cloning of data is necessary to adhere to the value semantics of Whiley. Clearly, if a large number of clones were required, then our approach could not compete with reference-based languages, such as Java.

5.3 Results

The results of our experiment are given in Table 1, which reports the ratio of update operations requiring a clone. A lower number indicates that more in place updates were performed. For completeness, we give the exact numbers of operations involved, the average reference count of an operand to any operation, and the average size (i.e. number of elements or fields) of cloned objects.

The results provide some interesting insights into the cost of using value semantics in Whiley. We note that, in most cases, the ratio of clone operations is extremely low. For example, looking at Gunzip, we see that only 873 out of 140561 operations could not be performed in place. For some benchmarks, *every single update operation was performed in place*. However, a number of benchmarks warrant further discussion:

- **Jasm** exhibits a high proportion of clone operations. A manual inspection reveals why: the inner loop decodes the bytecode at the current position in the byte list; it may then read a number of additional bytes (typically 1-4, for e.g. immediate values). The sublist operation extracts these additional bytes (e.g. `data[pos..pos+4]`. The byte list is live as it will be used in subsequent iterations and, hence, the sublist is not performed in place. Instead, a fresh list is created and those additional bytes are copied over. In this case, the average number of bytes copied is very small, and is not a cause of concern.

- **Queens** also exhibits a high proportion of clone operations. A manual inspection reveals why: the algorithm performs a brute force exploration where, at each stage, a queen is place on the board. The board is represented as a 2D list and, thus, each modification clones an inner list and the outermost list. This is also evident from the average clone size. In fact, a number of well-known optimisations of this algorithm have not been applied. Such optimisations would likely reduce the amount of cloning required.

- **SCC** exhibits a low proportion of clones, but a high average reference count for operands to update operations. This seems counter-intuitive, since in place updates require a reference count of one. A manual inspection did not reveal the reason. Instead, we examined the distribution of reference counts and made a surprising discovery: the vast majority of reference counts were one, but a *single* object with a high count (>500) was repeatedly updated. The algorithm requires four temporary lists which (for convenience) are packaged into a `State` record. The algorithm descends the graph and then, eventually, unwinds. During the first stage, the `State` object *is never directly updated*. However, as the algorithm unwinds *it is updated when components are found*. Thus, during the first stage, this object accumulates a very large number of references which are then encountered during unwinding.

In Table 1, we also report runtime with (and without) reference counting enabled. In reporting runtimes, our aim is only to provide assurance that our implementation is reasonably efficient. One cannot easily compare the performance of Whiley programs with e.g. Java, for several reasons:

- Whiley supports unbounded arithmetic. Thus, every integer in Whiley is implemented as a Java `BigInteger`. In the future, we hope to utilise native **`int`**s by exploiting integer range analysis.

- Our benchmark programs have not been written specifically for performance. For example, the Gunzip benchmark does not implement a number of optimisations found in the `gunzip` command-line tool.

- The code generated by the Whiley compiler it far from optimal, and significant work remains to be done there.

Furthermore, as noted above, our reference counting implementation is not particularly efficient, and could be further improved.

6 Related Work

OCaml [69] shares some interesting similarities with Whiley, as it supports structural typing and value semantics. Furthermore, mutable structures (e.g. arrays and mutable records) can be updated in place. Unlike Whiley, however, this is not done in accordance with value semantics — rather, mutable structures are allocated in the heap and passed by reference [70].

The Strongtalk system pioneered the use of structural typing to type structures in Smalltalk [71]. The essential idea is succinctly captured in the following quote (from [71]):

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

Unfortunately, structural subtyping was subsequently dropped from Strongtalk [72].

The work of Guha et al. focuses on flow-sensitive type checking for JavaScript [24]. 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* [23]. Their system retypes a variable within an expression dominated by a type test. They employ union types to increase the range of types, but fall short of full structural types. 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 [73]. They support more expressive types, but again do not consider recursive structural types. Following the soft typing discipline, runtime checks are inserted at points which cannot be shown type safe.

The Java Bytecode Verifier uses a flow-sensitive type checker [26]. Since locals and stack locations are untyped in Java Bytecode, it must infer their types to ensure type safety. 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 variable types after `instanceof` tests.

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 [12]. 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# [15]. 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 [13]. Pominville *et al.* also briefly discuss a flow-sensitive non-null analysis built using SOOT, which does retype variables after `v!=`**`null`** checks [21]. The JACK tool is similar, but focuses on bytecode verification instead [14]. JavaCOP provides an expressive language for writing type system extensions, including non-null types [22]. 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 `v` to a non-null variable if this is the first statement after a `v!=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 [17]. 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 [18]. 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 [16].

7 Conclusion

The Whiley language is based on updateable value semantics, supported by flow-sensitive and structural typing. This programming model enables flexible updates to compound structures and covariant subtyping of collections. We have formalised the type system using a core calculus called Featherweight Whiley (FW). We have proved soundness and termination properties for this system — full details can be found in [74]. We have also reported on experiments investigating the cost of value semantics with the conclusion that, in the majority of cases, update operations are performed in place. Finally, an open source implementation of Whiley is available from `http://whiley.org`.

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

References

[1] R. Cartwright and M. Fagan. Soft typing. In *Proceedings of the ACM conference on Programming Language Design and Implementation (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 *Proceedings of the Dynamic Languages Symposium (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 *Proceedings of the ACM conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 117–136, 2009.

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

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

[9] scala. The scala programming language. URL `http://lamp.epfl.ch/scala/`. http://lamp.epfl.ch/scala/.

[10] G. Bierman, E. Meijer, and M. Torgersen. Lost in translation: formalizing proposed extensions to C#. In *Proceedings of the ACM conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 479–498, 2007.

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

[12] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *Proceedings of the ACM conference on Programming Language Design and Implementation (PLDI)*, pages 1–12. ACM Press, 2002.

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

[14] C. Male, D. J. Pearce, A. Potanin, and C. Dymnikov. Java bytecode verification for @NonNull types. In *Proceedings of the confererence on Compiler Construction (CC)*, pages 229–244, 2008.

[15] M. Fähndrich and K. R. M. Leino. Declaring and checking non-null types in an object-oriented language. In *Proceedings of the ACM conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2003.

[16] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Proceedings of the ACM symposium on the Principles Of Programming Languages (POPL)*, pages 228–241, 1999.

[17] Sebastian Hunt and David Sands. On flow-sensitive security types. In *Proceedings of the ACM symposium on the Principles Of Programming Languages (POPL)*, pages 79–90. ACM Press, 2006.

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

[19] David J. Pearce. JPure: a modular purity system for Java. In *Proceedings of the confererence on Compiler Construction (CC)*, volume 6601 of *LNCS*, pages 104–123, 2011.

[20] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A theory of type qualifiers. In *Proceedings of the ACM conference on Programming Language Design and Implementation (PLDI)*, pages 192–203. ACM Press, 1999.

[21] P. Pominville, F. Qian, R. Vallée-Rai, L. Hendren, and C. Verbrugge. A framework for optimizing Java using attributes. In *Proceedings of the confererence on Compiler Construction (CC)*, 2001.

[22] C. Andreae, J. Noble, S. Markstrum, and T. Millstein. A framework for implementing pluggable type systems. In *Proceedings of the ACM conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2006.

[23] Sam Tobin-Hochstadt and Matthias Felleisen. Logical types for untyped languages. In *Proceedings of the ACM International Conference on Functional Programming (ICFP)*, pages 117–128, 2010.

[24] A. Guha, C. Saftoiu, and S. Krishnamurthi. Typing local control and state using flow analysis. In *Proceedings of the European Symposium on Programming (ESOP)*, pages 256–275, 2011.

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

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

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

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

[29] D. J. Pearce. Whiley: a language with flow-typing and updateable value semantics. Technical Report ECSTR12-09, Victoria University of Wellington, 2012.

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

[31] N. Lameed and L. Hendren. Staged static techniques to efficiently implement array copy semantics in a MATLAB JIT compiler. In *Proceedings of the confererence on Compiler Construction (CC)*, 2011.

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

[33] Martin Odersky. How to make destructive updates less destructive. In *Proceedings of the ACM symposium on the Principles Of Programming Languages (POPL)*, pages 25–36, 1991.

[34] C. Flanagan, K. R.M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the ACM conference on Programming Language Design and Implementation (PLDI)*, pages 234–245. ACM Press, 2002.

[35] M. Finifter, A. Mettler, N. Sastry, and D. Wagner. Verifiable functional purity in Java. In *Proc. CCS*, pages 161–174, 2008.

[36] Lars Ræder Clausen. A java bytecode optimizer using side-effect analysis. *Concur.-P&E*, 9(11): 1031–1045, 1997.

[37] Anatole Le, Ondrej Lhoták, and Laurie J. Hendren. Using inter-procedural side-effect information in JIT optimizations. In *Proceedings of the confererence on Compiler Construction (CC)*, 2005.

[38] D. J. Pearce. A calculus for constraint-based flow typing. In *Submitted to FTFJP*, 2012.

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

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

[41] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002. ISBN 0-262-16209-1.

[42] M. Cielecki, J. Fulara, K. Jakubczyk, and L. Jancewicz. Propagation of JML non-null annotations in Java programs. In *Proceedings of the conference on Principles and Practices of Programming in Java (PPPJ)*, pages 135–140, 2006.

[43] Patrice Chalin and Perry R. James. Non-null references by default in Java: Alleviating the nullity annotation burden. In *Proceedings of the European Confereince on Object-Oriented Programming (ECOOP)*, pages 227–247. Springer, 2007.

[44] Laurent Hubert. A non-null annotation inferencer for java bytecode. In *Proceedings of the Workshop on Program Analysis for Software Tools and Engineering*, pages 36–42. ACM, 2008. ISBN 978-1-60558-382-2.

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

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

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

[48] D. Malayeri and J. Aldrich. Integrating nominal and structural subtyping. In *Proceedings of the European Confereince on Object-Oriented Programming (ECOOP)*, pages 260–284, 2008.

[49] Luca Cardelli. Structural subtyping and the notion of power type. In *Proceedings of the ACM symposium on the Principles Of Programming Languages (POPL)*, pages 70–79. ACM Press, 1988.

[50] Portable network graphics (PNG): Functional specification. ISO/IEC 15948:2003 (E), 2003.

[51] ISO90. ISO/IEC. international standard ISO/IEC 9899, programming languages — C, 1990.

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

[53] P. Deutsch. DEFLATE compressed data format specification (RFC1951), 1996.

[54] Joshua Bloch. *Effective Java: Programming Language Guide*. Addison-Wesley, 2001.

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

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

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

[58] Castagna and Frisch. A gentle introduction to semantic subtyping. In *Proceedings of the ICALP*, pages 198–199, 2005.

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

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

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

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

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

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

[65] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark K. Wegman, and F. Kenneth Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the ACM symposium on the Principles Of Programming Languages (POPL)*, pages 25–35. ACM Press, 1989.

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

[67] Jong-Deok Choi, Vivek Sarkar, and Edith Schonberg. Incremental computation of static single assignment form. In *Proceedings of the confererence on Compiler Construction (CC)*, pages 223–237. Springer-Verlag, 1996.

[68] Richard E. Jones and Rafael Lins. *Garbage Collection*. Wiley, 1996.

[69] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon). *The Objective Caml system, Documentation and user's manual*, 2011.

[70] D Rémy. Using, understanding, and unraveling the OCaml language. from practice to theory and vice versa. In *Proc. APPSEM*, pages 413–536. Springer-Verlag, 2000.

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

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

[73] Alexander S. Aiken, Edward L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *Proceedings of the ACM symposium on the Principles Of Programming Languages (POPL)*, pages 163–173, 1994.

[74] D. J. Pearce and J. Noble. Whiley: a language combining flow-typing with updateable value semantics. Technical Report ECSTR12-10, Victoria University of Wellington, 2012.