Toward a Verified Software Toolchain for Java

David Pichardie

mémoire d'habilitation à diriger des recherches

défendu le 19 Novembre 2012
devant le jury composé de

MM	Chris	Hankin	Rapporteurs
	Tobias	Nipkow	
	Christine	Paulin-Mohring	
MM	Xavier	Leroy	Examinateurs
	Luc	Bougé	
	Thomas	Jensen	

Contents

Chapter 1

Introduction

Software are increasingly complex and are unavoidably subject to programming errors (*a.k.a.* bugs). The problem is well known and many techniques have been developed in order to reduce the number of bugs in a program. Among them, this document specially studies automatic verification techniques that operate at compile time and that aim at catching all errors of a certain kind: static analyses and type systems. For example, we can rely on an information flow type system to verify, before running or distributing a program, that it does not leak confidential information to the external environment.

One concern we can have is about the reliability of such a verification. Indeed, verification tools are themselves complex softwares. Moreover they make assumptions about the execution model of programs but this model is itself an abstraction of the real compiled code that is run at the end. Therefore the whole *software toolchain* of a programming language requires reliability[1]. Figure 1.1 sketches the various components of a software tool chain: starting from a source representation, a program is progressively transformed until it reaches a directly executable form. During these successive transformations, the program representation has to pass some verification checks (the type checking of the Java bytecode verifier at bytecode level for example). At some level, the current form of the program can be transformed for the purpose of verification only (form #4 in Figure 1.1). Transformation and verification phases all rely on a *semantic analysis* of the program that predict the dynamic behaviour of a piece of program without executing it.

Our ultimate objective is to build such a toolchain for the Java programming language. Our challenge here is to build a *verified* platform: each components (transformation, verification tools) should be formally specified in an expressive logic and correctness of the implementation of these components should be rigorously proved. Proof assistants are of special interest for these tasks: they provide a very rich specification language (based for example on high-order logic) with an automatic mechanism to check validity of proofs. The proof we are interested in are specially long and too error-prone to be fully verify at hand. Proof assistants allow for writing programs (here compilers and verification tools), their specification, and the corresponding correctness proof in a unified, logical framework. Several of them provide an extraction mechanism that automatically generates executable code that fulfills the formalized specification.

We mainly focus on Java because it is a modern language with several challenging features: security mechanisms, type and memory safety, modularity. Still many facets of our work are not fully specific to this programming language. The object-oriented programming paradigm is for example quite orthogonal in this work.

This document will summarise seven years of my research work around this objective. As we will see in conclusion the road is still long to achieve our goal but we already have learnt

[1]Hardware toolchains are equally important but out of the scope of this work.

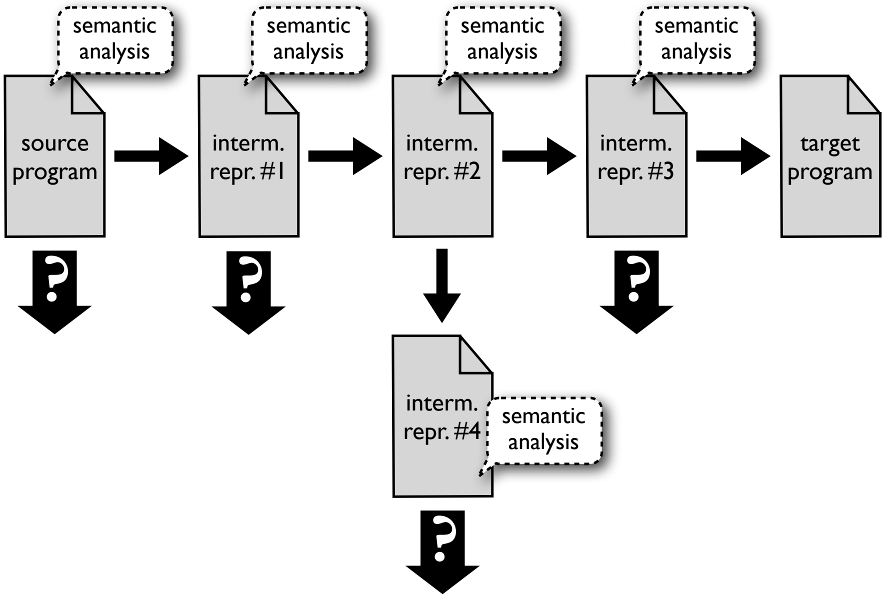


Figure 1.1: Typical toolchain components. A simple arrow represents a transformation. A large arrow with a question mark represents a verification.

some interesting lessons that we will share here.

1.1 Related Work

Verified programming language papers There is a recent interest in the programming language research area for research papers with companion machine checked proofs [ABF+05]. This phenomenon shows the maturity of proof assistants for this kind of exercises as well as the increasing quality of the tutorials and textbooks that are available to learn them. Most of the work I will report in this document lives however in an other category: we non only want to increase our confidence in the paper-proof of a verification technique but also in its implementation. For that purpose, we generally have to take into account more technical details (data-structures used in the implementation, interaction with all the other mechanisms of the language).

Verified verification tools Various verification techniques have been studied using machine checked proofs. Type systems as the Hindley-Milner type systems have been verified in the late 1990's by Nipkow in Isabelle [NN99] and by Dubois and Menissier-Morain in Coq [DMM99], at source level. At the machine code level, the Foundational proof-carrying code project of Appel *et al* was entirely formalized in LF [App01]. The Java bytecode verifier has been specially formalized in research projects such as Jinja [KN03], Jakarta [BCDM02], or in an industrial setting [C08]. Static analyses based on classic dataflow frameworks were mechanically verified by Nipkow [KN03], Coupet-Grimal and Delobel [CGD06], Bertot *et al* [BGL06], and Cachera *et al* [CJPR05]. Abstract Interpretation [CC77] provides an even more general framework for designing static analyses. My PhD work [Dav05] gave the first successful Coq mechanization of a full abstract interpreter. It provided a proof methodology to mechanise static analyses using the Abstract Interpretation framework while focusing on soundness. Bertot's course notes [Ber09a] give a tutorial about a verified abstract interpreter in Coq. Deductive program provers based on axiomatic semantics have also been formalised. Several authors have mechanized Hoare logic and its standard verification condition generator for toy imperative languages: see Gordon [M. 88], Nipkow [Nip98], Bertot [Ber09b], and Leroy [Ler10]. More realistic deductive verification tool techniques have been formalised for Java by Wildmoser and Nipkow [WN05] and Appel and Blazy for C [AB07]. Herms and al. [HMM12] recently provided a verified verification condition generator that follows closely the Why tool [FM07].

Verified Compilers Proving the correctness of compilers is not a new problem. The first such proof was mechanized as early as 1972 using the Stanford LCF proof assistant [MW72], based on the paper-proof by McCarthy and Painter [MP67]. Several projects have build verified compilers: J. Moore's Piton [Moo96], Guttman, Ramsdell and Wand's VLISP [GMR$^+$95], the German projects Verifix [GZ99] and Verisoft [P$^+$08], Klein and Nipkow's Jinja [KN06], Myreen's verified JIT compiler [Myr10] and the verified LLVM project by Zhao et al. [ZNMZ12]. The most advanced and realistic formally verified compiler today is the CompCert C compiler [Ler09a]. It handles a large subset of C, provides some carefully chosen optimizations – e.g. tail-call detection, constant propagation, common subexpression elimination with memory variables or function inlining – and produces assembly code for realistic processors (PowerPC, x86 and ARM).

1.2 Outline of the document

Verified Abstraction Interpretation My PhD's works dealt with the development of verified static analyses using the Coq proof assistant and the Abstract Interpretation theory. During this work, I have developed static analyses for a programming language similar to Java bytecode. The extraction features of Coq have enabled me to extract an analyser written in OCaml, from the correctness proof of the analysis. The correctness proof followed the abstract interpretation methodology. I proposed a generic framework to formalise static analysers in Coq.

Chapter 2 reports about two main extensions I did to this work after my PhD. The first one is a verified abstract interpreter for a simple imperative programming language. Using recent advances in the Coq proof assistant (type classes), David Cachera and I are able to program and prove correct an abstract interpreter with a complex iteration strategy [CP10] that follows the program syntax. The second extension addresses the problem of relational abstract domains as convex polyhedra [CH78]. We provide a verified result validator that is able to check the correctness result of such static analyses [BJPT10].

Toward Verified Static Analysis Tools for Java Bytecode Programs The purpose of this activity is to push further the scope of verified static analysis without special emphasis on abstract interpretation technique but trying to cope with more advanced semantic properties and more realistic features of modern programming languages. It is the purpose of Chapter 3.

The first semantic property I study is non-interference. Non-interference is a semantical condition on programs that guarantees the absence of illicit information flow throughout their executions, and that can be enforced by appropriate information flow type systems. Much of previous work on type systems for non-interference has focused on calculi or high-level programming languages, and existing type systems for low-level languages typically omit objects, exceptions, and method calls, and/or do not prove formally the soundness of the type system. Together with Gilles Barthe and Tamara Rezk we define an information flow type system for a sequential JVM-like language that includes classes, objects, arrays, exceptions and method calls, and prove that it guarantees non-interference [BPR07, BPR12]. For increased confidence, we formalise the proof in the Coq proof assistant.

The second challenging semantic property is data-race freeness. In multithreaded programming a data race occurs in a program execution when two threads access a memory location, and at least one of them changes its value, without proper synchronisation. Such situations can lead to unexpected behaviours, sometimes with damaging consequences. Formally ensuring data race freeness of a program is a central problem in Java, because the Java Memory Model then guarantees that one can safely reason on the interleaved semantics of the program. Together with Frédéric Dabrowski, we formalise in Coq a Java bytecode data race analyser [DP09] based on the conditional must-not alias analysis of Naik and Aiken [NAW06, NA07]. The formalisation includes a context-sensitive points-to analysis and an instrumented semantics that counts method calls and loop iterations. Our Java-like language handles objects, virtual method calls, thread spawning and lock and unlock operations for threads synchronisation.

I think these two works push to its limit what can be formally proved in a proof assistant with a direct approach where the proof and the implementation of the verified static analyser are performed directly at the level of the programming language (here the Java bytecode language). Each proof was more than 15.000 lines of Coq long and took more than one man-year. After this observation, it was clear to me that I need to build a suitable static analysis platform that will facilitate both the implementation and the proof of static analyser.

Foundations and Implementation of Static Analysis for Java Bytecode Programs Chapter 4 is closely related to Laurent Hubert's PhD thesis. Several static analysis platforms have been proposed in the past in order to facilitate the development of static analysis for Java bytecode programs. But none of them was designed with the goal of performing a soundness proof. The goal of this activity was hence to study several advanced static analyses and build an Ocaml library that makes possible both an efficient implementation and a high level reasoning. In Laurent Hubert's thesis several static analyses have been proposed to better master initialization in Java. Among them, a type system finely tracks initialization of objects and allows to enforce secure object initialization [HJMP10]. For both analyses, we formalise their semantic foundations, and prove their soundness in Coq. Furthermore, we also provide OCaml implementations that lead to realistic tools. During this PhD, and to ease the adaptation of such analyses, which have been formalized on idealized languages, to the full-featured Java bytecode, we have developed a OCaml library that gives a tool bench for static analysis of Java bytecode programs[2] (Sawja). After Laurent Hubert's work this approach has been pursued for a specific shape static analysis that verifies the correctness of Java copying methods [JKP11, JKP12], in collaboration with Florent Kirchner. The formal link between the idealized language and the full Java bytecode language has been formalized during the first part [DJP10] of Delphine Demange's PhD.

[2]http://sawja.inria.fr

Verified Static Single Assignment Intermediate Representation The previous activity has built the foundations of a tool bench for developing provably correct and realistic static analysers for the Java bytecode language. The Sawja program representations can be quite easily lift from Ocaml structures to Coq structures. But it appears that giving a realistic semantics to this representation is still difficult since, as many modern compiler and static analysis platforms, Sawja provides a SSA representation. In Chapter 5 we study this intermediate representation [BDP12]. We build a new SSA middle-end on top of the CompCert compiler. For the proof of SSA optimizations, we identify a key semantic property of SSA, allowing for equational reasoning. This work is based on Delphine Demange's PhD.

Remaining steps for a verified Java toolchain Among the remaining steps to reach our ultimate objective we identify concurrency as a clear bottleneck. Chapter 6 concludes this document by explaining the related issued and our ongoing works in this area.

Chapter 2

Verified Abstraction Interpretation

A verified static analysis is an analysis whose semantic validity has been formally proved correct with a proof assistant[1]. The main contribution of my PhD thesis was the development of machine checked proofs of soundness of static analyses using the general theoretical framework of Abstract Interpretation. A general methodology has been designed to formalise static analyses in the specific logic of the Coq proof assistant. The key feature was to isolate the parts of the theoretical Abstract Interpretation framework that were mandatory in a soundness proof. Together with the methodology, a general library has been developed to build complex lattices in Coq together with a proof of termination of generic fixpoint iterations on these lattices. Such a machine-checked library was the first to go beyond the notion of ascending chain condition and to rely instead on the more general notion of widening/narrowing operators. During this PhD, several instantiations of the framework were developed for the static analysis of bytecode programs.

This chapter summarises my activities in this area after my PhD. The area of Abstraction Interpretation is deep and several of its specific techniques were still interesting to study in the context of verified static analyses. Among them, my colleagues and I have studied more closely the notions of collecting semantics and relational abstract domains.

2.1 Denotational Abstract Interpreter

We follow in this section the paper published in [CP10]. This section will expose far more Coq syntax than the other chapters of the document but most of the current contribution is not *what* we formalise but *how* we achieve the formalisation.

The technique we formalise solves two issues.

Collecting Semantics First, instead of directly proving the abstract interpreter with respect to an operational semantics, we introduce an important semantic step for the Abstract Interpretation methodology: the *collecting semantics*. Usually, I present the notion of collecting semantics to students in the following terms: *a programming language semantics that looks like a static analysis but that is concrete enough to be still called a semantics.* Indeed, the purpose of a collecting semantics is to describe mathematically the precise semantic information that we focus in a given static analysis. Its form must be close enough to the static analysis itself, to make parts of the static analysis soundness proof quite obvious. In turn, its definition must be equivalent to a more standard semantics but this step is often skipped in

[1]I have often used the term *certified* in my own works but I think the term is somewhat confusing with respect to the software *certification* process that is used in the safety critical industry.

the literature. As far as we know, this is the first time the slogan "my abstract interpreter is correct by construction" is turned into a precise machine-checked proof.

Denotational Fixpoint Iteration The second issue is related to fixpoint iterations on equations systems. Abstract interpreters generally require to find a solution $(X_1, \ldots, X_n)$ to a *post-fixpoint* problem of the form

$$\begin{array}{rcl} X_1 & \sqsupseteq & F_1(X_1, \ldots, X_n) \\ & & \ldots \\ X_n & \sqsupseteq & F_n(X_1, \ldots, X_n) \end{array}$$

where each X_i lives in a lattice structure $(A, \sqsupseteq, \sqcup, \sqcap, \bot, \top)$.

Starting from a pre-fixpoint (generally the bottom element $(\bot, \ldots, \bot)$), one can iteratively update each components by choosing one equation F_i and replace the current value of X_i by $X_i \sqcup F_i(X_1, \ldots, X_n)$, until one reach a solution. To ensure convergence (and also to accelerate convergence on lattices with large heights), one can use a widening operator instead of the least upper bound $\sqcup$. In practice, if one use such a widening operator on each components of the equation system, the final post-fixpoint is very unprecise (*i.e.* it does not give the program invariants one would like to obtain while their exist other post-fixpoints that do).

The first optimisation to perform is to restrain the usage of widening on a restricted subset of components (we will call this set the *widening points*). To preserve termination, every dependency loop in the dependency graphs of the equation system must contain a widening point.

The second optimisation deals with the order in which one choose equations. In a standard monotone framework where operators F_i are monotone and where lattices do not contain infinite ascending chains, this order does not matter: the first fixpoint one reach is always the least post-fixpoint (which coincides with the least fixpoint of the system). In a more general setting, widenings are non-monotone [CC92] and the precision of the final post-fixpoint depends directly on the order in which one choose equations during the iteration process. One good strategy here is to ensure convergence for the most inner loops (with respect to the equation dependency graph) first and progressively propagate this information to the outer loops.

For structured programs, these two optimisations can be obtained by building a fixpoint iterator that iterates directly on the program syntax [Cou99]. The widening operator is only used when interpreting a loop construction and for each instruction, one recursively interpret instruction sub-terms (as loop body) before interpreting the rest of the instruction. The interpreter we present in this section follows this design.

2.1.1 Syntax and Standard Semantics

The starting point of the development is the formal syntax of a classical imperative structured language.

```
Definition pp := word.

Definition var := word.

[...]
```

```
Inductive instr :=
   | Skip (p:pp)
   | Assign (p:pp) (x:var) (e:expr)
   | Assert (p:pp) (t:test)
   | If (p:pp) (t:test) (b1 b2:instr)
   | While (p:pp) (t:test) (b:instr)
   | Seq (i1:instr) (i2:instr).

Record program := { p_instr:instr;  p_end: pp;  vars: list var }.
```

The Coq keyword **Inductive** is used here to define a recursive datatype.

Programs are labelled with elements of type `pp := word`, which plays a special role in all our development. It is the type of Coq binary numbers with at most 32 bits and is hence inhabited with a finite number of objects. This property is crucial in order to ensure the termination of the fixpoint iteration on arrays indexed by keys of type `word`. A program is made from a main instruction, a end label and a set of local variables. Instructions contain skip, assignment of a variable by a numeric expression, conditional, while loop and sequence. The `Assert` instruction is only executable when the given assertion holds. The execution is otherwise blocked. The syntax of expressions and tests is standard and skipped here.

We then give to this language a straightforward small-step operational semantics.

```
Definition env := var → Z.
Inductive config := Final (ρ:env) | Inter (i:instr) (ρ:env).
```

An environment (of type `env`) maps variable names to numerical values. A configuration is either a final environment or an intermediate configuration. The operational semantics takes the form of a judgment `(sos p (i,ρ) s)` that reads as follows[2]: for a program `p` there is a one-step transition between an intermediate configuration `(i,ρ)` (an instruction and an environment) towards configuration `s`.

```
Inductive sos (p:program) : (instr*env) → config → Prop :=
| sos_assign : ∀ l x e n ρ1 ρ2,
  sem_expr p ρ1 e n → subst ρ1 x n ρ2 → In x (vars p) →
  sos p (Assign l x e,ρ1) (Final ρ2)
[...]
| sos_while_true : ∀ l t b ρ,
  sem_test p ρ t true →
  sos p (While l t b,ρ) (Inter (Seq b (While l t b)) ρ)
| sos_while_false : ∀ l t b ρ,
  sem_test p ρ t false →
  sos p (While l t b,ρ) (Final ρ)
```

Where predicate `(subst ρ1 x n ρ2)` expresses that the environment `ρ2` is the result of the substitution of `x` by `n` in `ρ1`. In a formula, → can be interpreted as an implication connector.

We do not detail the bigstep evaluation judgments for expression and tests (`sem_expr` and `sem_test`). We use this time the Coq keyword **Inductive** to introduce an inductive relation. It is written with the following pattern

```
Inductive R : A → B → Prop :=
 | Rule1: ∀ a b, ... → R a b
 | Rule2:. .. → R a b
```

[2]The definition given in [CP10] provides an extra argument that records the kind of transition that is taken. It is a technical trick that is very useful in the soundness proof of the collecting semantics but we do not expose it here.

meaning that the relation `R` is a binary predicate (indicated by **`Prop`**, the type of propositions in Coq) whose arguments are of types `A` and `B` respectively. The relation `R` is defined by two rules `Rules1` and `Rules2`, describing when the proposition `(R a b)` holds for elements `a` and `b` (the hypotheses are indicated by dots). In the definition of `sem`, the variable `p` is a *parameter*, *i.e.* a fixed argument of the inductive relation.

The soundness theorem of the analyser is expressed in terms of labelled reachable states in `(pp * env)`. The special label `(p_end p)` is attached to final environments. The function `first` recursively computes the left-most label of an instruction.

```
Inductive reachable_sos (p:program) : pp*env → Prop :=
| reachable_sos_intermediate : ∀ ρ0 i ρ,
  sos_star p (p_instr p,ρ0) (Inter  i ρ) → reachable_sos p (first i,ρ)
| reachable_sos_final : ∀ ρ0 ρ,
  sos_star p (p_instr p,ρ0) (Final ρ) → reachable_sos p (p_end p,ρ).
```

Note that we take as initial environments, any `ρ0`. It can be read as an existentially quantified variable since rule `reachable_sos_intermediate` is equivalent to

```
  ∀ i ρ,
   (∃ ρ0, sos_star p (p_instr p,ρ0) (Inter  i ρ))  →  reachable_sos p (first i,ρ)
```

The reflexive transitive closure `sos_star` of `sos` is defined in a straightforward way.

2.1.2 Lattice Theory Intermezzo

The collecting semantics we will define in the next section will rely on a generic least fixpoint operator over complete lattices. We describe in this section the signatures of these mathematical components. The following declarations define the notion of equivalence relation, partial order, subsets and complete lattices.

```
Module Equiv.
  Class t (A:Type) : Type :=
   { eq : A → A → Prop;
     refl : ∀ x, eq x x; sym : [...]; trans : [...] }.
End Equiv.
Notation "x == y" := (Equiv.eq x y) (at level 40).

Module Poset.
  Class t A : Type :=
   { eq :> Equiv.t A;
     order : A → A → Prop;
     refl : [...];  antisym : [...]; trans : [...] }.
End Poset.
Notation "x ⊑ y" := (Poset.order x y) (at level 40).

Class subset A {Equiv.t A} : Type := SubSet {
  charact : A → Prop;
  subset_comp_eq : ∀ x y:A, x==y →  charact x → charact y}.
Coercion charact : subset >-> Funclass.

Module CompleteLattice.
  Class t (A:Type) : Type := {
    porder :> Poset.t A;
    join : subset A → A;
    join_bound : ∀x:A, ∀f:subset A, f x → x ⊑ join f;
    join_lub : ∀f:subset A, ∀z, (∀ x:A, f x → x ⊑ z) → join f ⊑ z;}.
End CompleteLattice.
```

Each **Class** keyword introduces a record that mixes declarations of several natures: predicates (as `eq:A→A→Prop`), functions (as `join : subset A →A`) and properties about them (as `refl : ∀x, eq x x`). For example, an object of type `Equiv.t nat` will handle a binary predicate of type `nat→nat→Prop` on natural numbers, plus a proof that this relation is reflexive, symmetric and transitive. We use here type classes that give elegant notations for defining Coq records with implicit arguments facilities. Modules are just used for the purpose of space naming. Coercions are also introduced in order to define `(Equiv.t A)` as subclass of `(Poset.t A)` (notation `:>` in the field `eq` of the class type `Poset.t`) and to view subset components of type `subset A` as simple predicates on `A` (command **`Coercion`** `charact : subset >->`**`Funclass`**). Some parameters are given with curly braces `{...}` in order to declare them as implicit. All these features make the formal definition far more elegant and concise. For example, in the type declaration of the field `join_bound` of `CompleteLattice.t`, the term `subset A` hides an object `(Poset.eq A porder)` of type `(Equiv.t A)` which is automatically taken from the field `porder : Poset.t A` of the class type `CompleteLattice.t` and the coercion between `(Equiv.t A)` and `(Poset.t A)`. Without the power of type classes inference, we could not use notations like $\sqsubseteq$ (introduced with the keyword **`Notation`**) since the Coq system would be unable to discover on its own which poset structure we are considering. For comparison, an equivalent definition of complete lattices with a simple record would looks like the following:

```
Module CompleteLattice.
  Record t (A:Type) : Type := Make
    { porder :> Poset.t A;
      join : subset A (Poset.eq porder) → A;
      join_bound : ∀x:A, ∀f: subset A (Poset.eq porder),
        f x → (Poset.order porder x (join f));
      join_lub : ∀f: subset (Poset.eq porder),
         ∀z, (∀ x:A, f x → (Poset.order porder x z)) →
         Poset.order porder (join f) z
    }.
End CompleteLattice.
```

We also introduce definitions for lattices (type class `Lattice.t`) with the corresponding overloaded symbols $\bot$, $\top$, $\sqcap$, $\sqcup$ but do not show them here to keep the presentation short.

Canonical structures have been defined for these types.

```
Notation "'𝒫' A" := (A → Prop) (at level 10).
Instance PowerSetCL A : CompleteLattice.t (𝒫 A) :=
  [...]

Instance PointwiseCL A L {CompleteLattice.t L} : CompleteLattice.t (A → L) :=
  [...]
```

The keyword **`Instance`** informs the Coq system that this definitions can be automatically introduced by the inference system when objects of these types will be wanted by the type checker.

We finish this section with the construction of the least-fixpoint operator. We first introduce the type of monotone functions using a dependent pair and a coercion.

```
Class monotone A {Poset.t A} B {Poset.t B} : Type := Mono {
  mon_func : A → B;
  mon_prop : ∀ a1 a2, a1 ⊑ a2 → (mon_func a1) ⊑ (mon_func a2)}.
Coercion mon_func : monotone >-> Funclass.
```

We conclude using the standard Knaster-Tarski theorem.

```
Definition lfp {L} {CompleteLattice.t L} (f:monotone L L) : L :=
  CompleteLattice.meet (PostFix f).

Section KnasterTarski.
  Variable L : Type.
  Variable CL : CompleteLattice.t L.
  Variable f : monotone L L.
  Lemma lfp_fixpoint : f (lfp f) == lfp f. [...]
  Lemma lfp_least_fixpoint : ∀ x, f x == x → lfp f ⊑ x. [...]
  Lemma lfp_postfixpoint : f (lfp f) ⊑ lfp f. [...]
  Lemma lfp_least_postfixpoint : ∀x, f x ⊑ x → lfp f ⊑ x. [...]
End KnasterTarski.
```

A Coq **Section** is used to pack four lemmas that share the three universally quantified arguments `L`, `CL` and `f`. We make use of a greatest lower bound `CompleteLattice.meet` of type `subset L (Poset.eq porder) →L` that is defined using the least upper bound operator. `(PostFix f)` is defined a the subset of all post-fixpoints of `f`.

2.1.3 An Intermediate Collecting Semantics

The collecting semantics assigns to all programs a map $\mathcal{C}$ of type `pp →`$\mathcal{P}$`(env)` such that $\mathcal{C}$`(p)` contains the set of environments that can be reached at program point `p`. We could define this semantics in a direct way with the previous `reachable_sos` predicate:

```
𝒞 pp env := reachable_sos prog (pp,env)
```

Still, it would not fit our purpose. We want to characterise this semantics in a way that makes it look closer to the final abstract interpreter. In particular, the fixpoint scheme should be similar. As we want to program and verify an abstract interpreter that inductively follows the syntax of programs and iterates each loop until convergence, we will introduce a collecting semantics that follows the same strategy, but computes on concrete properties.

In all this section we fix a program `prog:program`. We first introduce three semantic operators. Operator `assign` is the strongest post-condition of the assignment of a variable by an expression. Operator `assert` is the strongest post-condition of a test. An element in `pp →`$\mathcal{P}$`(env)` can be updated with the function `Esubst`. It is called a weak update since we take the union with the previous value.

```
Definition assign (x:var) (e:expr) (E:𝒫(env)) : 𝒫(env) :=
  fun ρ => ∃ρ', ∃n, E ρ' ∧ sem_expr prog ρ' e n ∧ subst ρ' x n ρ.
Definition assert (t:test) (E:𝒫(env)) : 𝒫(env) :=
  fun ρ => E ρ ∧ sem_test prog ρ t true.
Definition Esubst (f:pp → 𝒫(A)) (k:pp) (v:𝒫(A)) : pp → 𝒫(A) :=
  fun k' => if pp_eq k' k then (f k) ⊔ v else f k'.
Notation "f +[ x ↦ v ]" := (Esubst f x v) (at level 100).
```

The collecting semantics is then defined by recursion on the program syntax. The expression `(Collect i l)` computes for an instruction `i` and a label `l`, a monotone predicate transformer such that for each initial property `Env:`$\mathcal{P}$`(env)`, all states `(l',ρ)` that are reachable by the execution of `i` from a state in `Env`, satisfy `(Collect i l Env l' ρ)` where `l'` is either a label in `i` or is equal to `l`. The label `l` should represent the next label after `i` in the whole program. The monotony property of the predicate transformer returned by `(Collect i l)` is crucial if we want to be able to use the `lfp` operator of the previous section that takes only monotone functions as argument. For each instruction, we hence build a term of the form `(Mono F _)` with `F` a function and `_` a "hole" for the proof that ensures the monotony of `F`. All these holes give rise to proof obligations that are generated by the **Program** mechanism and must be interactively discharged after the definition of `Collect`.

```
Program Fixpoint Collect (i:instr) (l:pp): monotone (𝒫(env)) (pp → 𝒫(env)) :=
match i with
  | Skip p => Mono (fun Env => ⊥ +[p ↦ Env] +[l ↦ Env]) _
  | Assign p x e =>
    Mono (fun Env => ⊥ +[p ↦ Env] +[l ↦ assign x e Env]) _
  | Assert p t =>
    Mono (fun Env => ⊥ +[p ↦ Env] +[l ↦ assert t Env]) _
  | If p t i1 i2 =>
    Mono (fun Env =>
          let C1 := Collect i1 l (assert t Env) in
          let C2 := Collect i2 l (assert (Not t) Env) in
            (C1 ⊔ C2) +[p ↦ Env]) _
  | While p t i =>
    Mono (fun Env =>
          let I:𝒫(env) := lfp (iter Env (Collect i p) t p) in
            (Collect i p (assert t I)) +[p ↦ I] +[l ↦ assert (Not t) I]) _
  | Seq i1 i2 =>
    Mono (fun Env => let C := (Collect i1 (first i2) Env) in
                         C ⊔ (Collect i2 l (C (first i2)))) _
end.
```

The `While` instruction is the most complex case. It requires to compute a predicate `I:𝒫(env)` that is a loop invariant bound to the label `p`. It is the least fixpoint of the monotone operator `iter` defined below.

```
Program Definition iter (Env:𝒫(env)) (F:monotone (𝒫(env)) (pp → 𝒫(env)))
     (t:test) (l:pp) : monotone (𝒫(env)) (𝒫(env)) :=
  (Mono (fun X => Env ⊔ (F (assert t X) l)) _).
```

We hence compute the least fixpoint `I` of the following equation:

```
                I == Env ⊔ (Collect i p (assert t I) p)
```

The invariant is the union of the initial environment (at the entry of the loop) and the result of the transformation of `I` by two predicate transformers: `assert t` takes into account the entry in the loop, and the recursive call to `(Collect i p)` collects all the reachable environments during the execution of the loop body `i` and select those that are bound to the label `p` at the loop header.

Such a semantics is sometimes taken as *standard* in Abstract Interpretation works but here, we precisely prove its relationship with the more standard semantics of Section 2.1.1.

```
Theorem sos_star_implies_Collect : ∀ i1 ρ1 s2,
  sos_star prog (i1,ρ1) s2 →
    match s2 with
      | Inter i2 ρ2 => ∀l:pp, ∀Env:𝒫(env),
          Env ρ1 → Collect i1 l Env (first i2) ρ2
      | Final ρ2 => ∀l:pp, ∀Env:𝒫(env),
          Env ρ1 → Collect i1 l Env l ρ2
    end.
```

A proof sketch of this theorem is given in [CP10]. It is now easy to deduce that each reachable state in the standard semantics is reachable with respect to the collecting semantics.

```
Definition reachable_collect (p:program) (s:pp*env) : Prop :=
  let (k,env) := s in Collect p (p_instr p) (p_end p) (⊤) k env.
Theorem reachable_sos_implies_reachable_collect :
```

```
∀ p s, reachable_sos p s → reachable_collect p s.
```

Note that the reverse inclusion

```
∀ p s, reachable_collect p s → reachable_sos p s.
```

is expected but not formally proved here. Indeed, proving such a result would require to add some assumptions on program points. From now, we have not required them to be unique (*i.e.* a same program point does not appear twice in a program). The result can be shown under this extra hypothesis but it is not strictly mandatory for the final theorem of this development.

2.1.4 A Verified Abstract Interpreter

We now design an abstract interpreter that, instead of executing a program on environment properties as the previous collecting semantics does, computes on an abstract domain with a lattice structure. Such a structure is modelled with a type class `AbLattice.t` that packs the standard components of lattices but also decidable tests for equality and partial order, and widening/narrowing operators. It is equipped with notations $\sqsubseteq^\sharp$, $\sqcap^\sharp$, $\sqcup^\sharp$, $\bot^\sharp$. This abstract lattice structure is a direct adaptation of our earlier work with Coq modules [Pic08]. The lattice signature contains a well-foundedness proof obligation which ensures termination of a generic post-fixpoint iteration algorithm.

```
Definition approx_lfp : ∀ {t} {AbLattice.t t}, (t → t) → t := [...]
Lemma approx_lfp_is_postfixpoint : ∀ t {AbLattice.t t} (f:t → t),
    f (approx_lfp f) ⊑♯ (approx_lfp f).
```

A library is provided to build complex lattice objects with various functors for products, sums, lists and arrays [Pic08]. Arrays are defined by means of binary trees whose keys are elements of type `word`. The corresponding functor `ArrayLattice` given below relies on the finiteness of `word` to prove the convergence of the pointwise widening/narrowing operators on arrays.

```
Instance ArrayLattice t {L:AbLattice.t t}: AbLattice.t (array t).
```

We connect concrete and abstract lattices with concretization functions that enjoy a monotony property together with a meet morphism property. It corresponds to the following type class `Gamma.t`.

```
Module Gamma.
  Class t a A {Lattice.t a} {AbLattice.t A} : Type := {
    γ : A → a;
    γ_monotone : ∀ N1 N2:A, N1 ⊑♯ N2 → γ N1 ⊑ γ N2;
    γ_meet_morph : ∀ N1 N2:A, γ N1 ⊓ γ N2 ⊑ γ (N1 ⊓♯ N2)
  }.
End Gamma.
Coercion Gamma.γ : Gamma.t >-> Funclass.
```

This formalization choice is the result of my PhD: one only keep the Abstract Interpretation elements that are necessary to perform a soundness proof and that fit well in the constructive logic of Coq [Dav05].

Using the new Coq type class feature we have extended our previous lattice library with *concretization functors* in order to build concretization operators in a modular fashion. We show below the signature of the array functor. This kind of construction was difficult with Coq modules that are not first class citizens, whereas it is often necessary to let concretizations depend on other terms, *e.g.* While programs.

```
Instance GammaFunc a A {Gamma.t a A} : Gamma.t (word → a) (array A) := [...]
```

Our abstract interpreter is parametrized with respect to an abstraction of program environments given below. The structure encloses a concretization operator mapping environment properties to an abstract lattice, two correct approximations of the predicate transformers `Collect.assign` and `Collect.assert` defined in Section 2.1.3, and an approximation of the "don't know" predicate (⊤).

```
Module AbEnv.
Class t {A} (L:AbLattice.t A) (p:program) : Type := {
  gamma :> Gamma.t (𝒫 env) A;
  assign : var → expr → A → A;
  assign_correct : ∀ x e, (Collect.assign p x e) ∘ gamma ⊑ gamma ∘ (assign x e);
  assert : test → A → A;
  assert_correct : ∀ t, (Collect.assert p t) ∘ gamma ⊑ gamma ∘ (assert t);
  top : A;
  top_correct : ⊤ ⊑ gamma top
}.
End AbEnv.
```

The abstract interpreter `AbSem` is then defined in a Coq section where we fix a program and an abstraction of program environments. Its definition perfectly mimics the collecting semantics. We use the abstract counterpart `F +[p ↦Env]♯` of the operator `Esubst` that has been defined in Section 2.1.3. The main difference is found for the `While` instruction where we do not use a least fixpoint operator but the generic post-fixpoint solver `approx_lfp`.

```
Section AbstractInterpreter.
  Variable (t : Type) (L : AbLattice.t t) (prog : program) (Ab : AbEnv.t L prog).

  Fixpoint AbSem (i:instr) (l:pp) : t → array t :=
    match i with
     | Skip p => fun Env => ⊥♯ +[p ↦ Env]♯ +[l ↦ Env]♯
     | Assign p x e =>
       fun Env => ⊥♯ +[p ↦ Env]♯ +[l ↦ Ab.assign Env x e]♯
     | Assert p t =>
       fun Env => ⊥♯ +[p ↦ Env]♯ +[l ↦ Ab.assert t Env]♯
     | If p t i1 i2 => fun Env =>
             let C1 := AbSem i1 l (Ab.assert t Env) in
             let C2 := AbSem i2 l (Ab.assert (Not t) Env) in
                 (C1 ⊔♯ C2) +[p ↦ Env]♯
     | While p t i => fun Env =>
       let I := approx_lfp
                   (fun X => Env ⊔♯ (get (AbSem i p (Ab.assert t X)) p)) in
         (AbSem i p (Ab.assert t I)) +[p ↦ I]♯ +[l ↦ Ab.assert (Not t) I]♯
     | Seq i1 i2 =>  fun Env =>
       let C := (AbSem i1 (first i2) Env) in
         C ⊔♯ (AbSem i2 l (get C (first i2)))
    end.
End AbstractInterpreter.
```

This abstract semantics is then proved correct with respect to the collecting semantics `Collect` and the canonical concretization operator on arrays. The proof is particularly easy because `Collect` and `AbSem` share the same shape.

```
Definition γ : Gamma.t (word → 𝒫(env)) (array t) := GammaFunc (word → 𝒫(env)) (array t).

Lemma AbSem_correct : ∀ i l_end Env,
  Collect prog i l_end (Ab.γ Env) ⊑ γ (AbSem i l_end Env).
```

At last we define the program analyser and prove that it computes an over-approximation of the reachable states of a program. This is a direct consequence of the previous lemma. Note that this final theorem deals with the standard operational semantics proposed in Section 2.1.1. The collecting semantics in only used as an intermediate step in the proof.

```
Definition analyse : array t := AbSem prog.(p_instr) prog.(p_end) (Ab.top).
Theorem analyse_correct : ∀ k ρ,
  reachable_sos prog (k,ρ) → Ab.γ (get analyse k) ρ.
```

In order to instantiate the environment abstraction, we provide in [CP10] a functor that builds a non-relational abstraction from any numerical abstraction by binding a numerical abstraction to each program variable.

```
Instance EnvNotRelational {L} (NumAbstraction.t L) (p:program) :
    AbEnv.t (ArrayLattice L) p := [...]
```

We do not detail here the content of the type `NumAbstraction.t` to keep the presentation short. We have instantiated it with interval, sign and congruence abstractions. The different instances of the analyser can be extracted to OCaml code and run on program examples. The Coq development is available at

http://irisa.fr/celtique/pichardie/ext/itp10

2.2 Verified Relational Abstraction by Result Validation

The goal of this section is to extend the previous formalisation with relational abstractions for environment. Instead of computing for each variable a property that holds for the values of this variable (*e.g.* $x = 0 \wedge y \geq 0$), we compute a relation between the values of all theses variables (*e.g.* $x < y$). The most well known relational abstract domains are convex polyhedra [CH78] and octagons [Min06b]. Those domains rely on highly engineered data structured to ensure scalability. They require fined tuned widening/narrowing in order to reach a good precision during loop invariant inference. For polyhedra, directly implementing in Coq the previous `AbEnv.t` interface would be a major *tour de force*. In this section we provide an alternative to this proof effort. First we relax the requirements of the previous interfaces while maintaining the soundness of the analyser. Then we show that, we can still use an efficient implementation of the relational abstract domain without loosing a formal proof of soundness of the analyser: the key technique is *a posteriori* validation.

The content of this section is based on [BJPT10] but we have completely revised its presentation. The work in [BJPT10] illustrates our technique on a polyhedral array-bound analysis for an imperative, stack-oriented bytecode language with procedures, arrays and global variables. Here, we keep the imperative language that we have used in this whole section. The corresponding formal development is available online at

http://irisa.fr/celtique/pichardie/ext/polycert

2.2.1 A Weaker Contract for Environment Abstraction

The previous signature for environment abstraction requires more proof that what is strictly necessary to prove soundness. During my PhD work I have isolated an Abstract Interpretation framework that was sufficient to prove soundness of various static analyses but the discussion was focused around the notion of abstraction that are either expressed by abstraction functions, concretisations function, or both (Galois connections). The notion of lattice was taken as granted during this work but as shown by Cousot et al [CCF+07] we dot not even need all the lattice properties.

The following signature relaxes the previous requirements.

```
Module AbEnv'.
Class t (A : Type) := Make {
  order :> A → A → bool;
  γ :> A → (𝒫 env);
  monotone : ∀ a1 a2:A, (order a1 a2 = true) → (γ a1) ⊑ (γ a2);

  bot :> A;
  bot_empty : (γ bot) ⊑ (fun _ => False);

  join :> A → A → A;
  join_bound1 : ∀ a1 a2:A, (γ a1) ⊑ (γ (join a1 a2));
  join_bound2 : ∀ a1 a2:A, (γ a2) ⊑ (γ (join a1 a2));

  assign : var → expr → A → A;
  assign_correct : ∀ x e, (Collect.assign x e) ∘ γ ⊑ γ ∘ (assign x e);

  assert : expr → A → A;
  assert_correct : ∀ t, (Collect.assert t) ∘ γ ⊑ γ ∘ (assert t);

  top : A;
  top_correct : ⊤ ⊑ γ top;

  widen :> A → A → A;
}.
End AbEnv'.
```

Now, lattice elements are given without proofs anymore. We only require γ to be monotone and we only consider the executable version of the abstract partial order test. We do not prove that is represents indeed a partial order. `bot_empty` requires `(γ bot)` to be the empty set. It is still not strictly necessary for soundness but useful in practice. We use this hypothesis in order to lazily represent arrays of abstract values. All other requirements are just the usual abstract interpretation correctness criteria of the form $F \circ \gamma \sqsubseteq \gamma \circ F^\sharp$. The least upper bound operator is handled in a similar way without requiring that it is indeed a minimal upper bound. Such an hypothesis is only mandatory to prove that we can obtain a least fixpoint with classical Kleene iteration but since we are using widening/narrowing, this property is already lost.

Using this contract, we program and prove sound a similar abstract interpreter as before. The novelty appears when analysing loop statements: the `approx_lfp` we use to compute abstract post-fixpoint follows a very similar algorithm as before but does not require termination proof for widenings. As it is done in Compcert [Ler06], we just fix a large natural number that explicitly bounds the number of steps we want to use when building our increasing iteration sequence. We return `top` if a post-fixpoint has not been reached before this maximum number of steps. In practice, this limit is never reached if our widening is not bogus, but we do not have to formally prove this fact.

2.2.2 Result Validation with Verified Decision Procedures

In order to instantiate the previous contract, we design a verified result validator that is parameterized by an environment abstraction of the following signature.

```
Module ExternAbEnv.
  Class t A : Type := Make {
   to_formula : A → zformula;
   assign : var → expr → A → A;
   assume : expr → A → A;
   join : A → A → A;
   widen : A → A → A;
   top : A;
   bot : A
  }.
End ExternAbEnv.
```

The first thing to notice is the lack of any proof in the `ExternAbEnv.t` contract! Instantiating such a contract requires merely the same effort in Coq as in Ocaml. The other thing to notice is the addition of a new field `to_formula` that requires a translation from every abstract element to an arithmetic formula. Here the type `zformula` represents formula built with atomic formula of type `test` and propositional connector (disjunction and conjunction). In the rest of this section we note `ex_zformula` the type of formula built from formula of type `zformula` but prefixed with existential quantifiers.

Our goal now is to instantiate the signature `AbEnv'.t` that we have presented in Section 2.2.1 in order to build a functor of signature `(ExternAbEnv.t A) →(AbEnv.t' A)`. But we have to explain how the proof that are required in objects of type `AbEnv.t'` are automatically built.

Concretization In the rest of the section we note $[\![\texttt{F}]\!]\rho$ when an environment ρ is a model for the formula `F` (of type `zformula` or `ex_zformula`), *i.e.* $\rho \models \texttt{F}$. We then define the concretization of our environment abstraction as

```
  Definition gamma (ab:A) : (𝒫 env) :=  ⟦to_formula ab⟧
```

The purpose of our abstract interpreter is hence to generate valid assertions at each program point of a program.

Abstract partial order The signature `AbEnv'.t` also requires an abstract partial order test `order` such that `∀ a1 a2:A, (order a1 a2 = true) → (γ a1) ⊑ (γ a2)`. It means we need a possibly incomplete decision procedure on `zformula`. Coq has been providing for a long time a decision procedure for the quantifier-free fragment of Presburger Arithmetic but we need here a decision procedure that is both programmed and proved correct in Coq. This is called a reflexive tactic. My colleague Frédéric Besson has been specifically working on this task [Bes06] and I have recently assisted him for an extension to a richer logic [BCP11]. In this document we will assume that such a black box `zprover:zformula→zformula→bool` exists and verifies

```
∀ f1 f2: zformula, zprover f1 f2 = true → ⟦f1⟧ ⊑ ⟦f2⟧
```

The abstract partial order can hence be defined by

```
order a1 a2 := zprover (to_formula a1) (to_formula a2)
```

Transfert function The rest of the signature requires a sound abstract counterpart `F♯` for several concrete operators `F` (least upper bound, assignment, assertion and top). For each of them, we assume an abstract operator `F_extern` is given by the record of type `ExternAbEnv.t A` (but we don't have any proof in hand about this operator), and apply the following methodology:

1. First we program a sound counterpart $\texttt{F}_\mathbb{Z}$ in the abstract domain `zformula`. For example, if `F` is an unary operator, $\texttt{F}_\mathbb{Z}$ has type `zformula →ex_zformula` and must satisfy

   $$\forall\ \texttt{f:zformula},\ \texttt{F}([\![\texttt{f}]\!]) \sqsubseteq [\![\ \texttt{F}_\mathbb{Z}\ \texttt{(f)}\ ]\!]$$

   For each previous concrete operators, such an operator is easily defined:

   ```
              f1 ⊔ℤ f2 := Disjunction f1 f2
       (assignℤ x e) f := Exists(x',Conjunction f[x'/x] (Eq x e[x'/x]))
         (assertℤ t) f := Conjunction t f
                  topℤ := True
   ```

   For the assignment, the variable `x'` must be fresh in `f` and `e`. Note that we need existential quantifiers for the assignment operation and hence use the type `ex_zformula` for the codomain of $\texttt{F}_\mathbb{Z}$.

2. Then we program each `F♯` in the following manner.

   ```
   Definition F♯ (ab:A) : A :=
     let ab' := F_extern ab in
       if zprover (Fℤ (to_formula ab)) (to_formula ab') then ab' else top.
   ```

   For the particular case of `top` we perform a equality test to check that `to_formula top` is equal to $\texttt{top}_\mathbb{Z}$. Note that we use an extended version of `zprover` that has type `ex_zformula→zformula→bool` instead of `zformula→zformula→bool`. The extension is easily done since existential quantifiers in the left hand side of an implication can be turned into universal quantifiers at the top level.

3. Now, the correctness of `F♯` is easily shown. If the prover succeeds, the returned value `(F_extern ab)` (*a.k.a.* `ab'`) satisfies

   $$\begin{array}{rcll} \texttt{F}([\![\texttt{to\_formula ab}]\!]) & \sqsubseteq & [\![\ \texttt{F}_\mathbb{Z}\ \texttt{(to\_formula ab)}\ ]\!] & \textit{by correctness of } \texttt{F}_\mathbb{Z} \\ & \sqsubseteq & [\![\texttt{to\_formula (F\_extern ab)}]\!] & \textit{because the prover succeeded} \end{array}$$

   otherwise the returned value is `top` but `(`**`fun`**` _ => True)` $\sqsubseteq[\![\texttt{to\_formula top}]\!]$ holds. In all cases,

   $$\texttt{F}([\![\texttt{to\_formula ab}]\!]) \sqsubseteq [\![\texttt{to\_formula (F}^\sharp\ \texttt{ab)}]\!]$$

This ends the generic construction of an abstract environment from any external abstract environment of type `ExternAbEnv.t A`.

We have used this functor together with the OCaml binding of the Apron library [JM09] that provides an efficient implementation of octagons and convex polyhedra relational abstract domains. The resulting abstract interpreter can be extracted to OCaml code and run on program examples.

2.3 Related Work

The analyser we have formalized here is taken from lecture notes by Patrick Cousot [Cou99, Cou]. We follow only partly his methodology here. Like him, we rely on a collecting semantics which gives a suitable semantic counterpart to the abstract interpreter. This semantics requires elements of lattice theory that, as we have demonstrated, fit well in the Coq proof assistant. One first difference is that we do not take this collecting semantics as standard but formally link it with a standard small-step semantics. A second difference concerns the proof technique. Cousot strives to manipulate Galois connections, which are the standard abstract interpretation constructs used for designing abstract semantics. Given a concrete lattice of program properties and an abstract lattice (on which the analyser effectively computes), a pair of operators (α, γ) is introduced such that $\alpha(P)$ provides the best abstraction of a concrete property P, and $\gamma(P^\sharp)$ is the least upper bound of the concrete properties that can be soundly approximated by the abstract element $P^\sharp$. With such a framework it is possible to express the most precise correct abstract operator $f^\sharp = \alpha \circ f \circ \gamma$ for a concrete f. Patrick Cousot in another set of lecture notes [Cou99] performs a systematic derivation of abstract transfer functions from specifications of the form $\alpha \circ f \circ \gamma$. We did not follow this kind of symbolic manipulations here because they would require much proof effort: each manipulation of α requires to prove a property of optimal approximation. We restrict the current work to a soundness proof.

The While language has been the subject of several machine-checked semantics studies [Gor88, Nip98, Ber09a, Ler10] but few have studied the formalization of abstract interpretation techniques. A more recent approach in the field of semantics formalization is the work of Benton *et al.* [BKV09] which gives a Coq formalization of cpos and of the denotational semantics of a simple functional language. A first attempt of a certified abstract interpreter with widening/narrowing iteration techniques has been proposed in my PhD manuscript [Dav05]. In this previous work, the analyser was directly generating an (in)equation system that was solved with a naive round-robin strategy. The soundness proof was performed with respect to an ad hoc small-step semantics. Bertot [Ber09a] has formalized an abstract interpreter whose iteration strategy is similar to ours. His proof methodology differs from the traditional Abstract Interpretation approach since the analyser is proved correct with respect to a weakest-precondition calculus. The convergence technique is more ad hoc than the standard widening/narrowing approach we follow. We believe the inference capability of the abstract interpreters are similar but again, our approach follows more closely the Abstract Interpretation methodology with generic interfaces for abstraction environments and lattice structures with widening/narrowing. We hence demonstrate that the Coq proof assistant is able to follow more closely the textbook approach compare to Bertot's presentation.

Relational abstract domains have receive little attention so far in the area of machine-checked proof. The result validation technique we propose share some ideas with the work of Wildmoser *et al.* [WCN05] who certifies a VCGen that uses an untrusted interval analysis for producing invariants and that relies on Isabelle/HOL decision procedures to check the verification conditions generated with the help of these invariants.

2.4 Conclusions and Perspectives

We have presented a verified abstract interpreter for a While language which is able to automatically infer program invariants. We have in particular studied the syntax-directed iteration strategy that is used in the Astrée tool. A similar interpreter had been proposed earlier [Ber09a] but our approach follows more closely the Abstract Interpretation methodology. The key ingredient of the formalization is an intermediate collecting semantics which is proved conservative with respect to a classical structural operational semantics.

The current work is a first step towards a global objective of putting in the Coq proof assistant most of the advanced static analysis techniques that are used in an analyser like Astrée. First we would need to enhance the current work with function calls in the language. That step should be easy to handle if we avoid recursion: as our abstract interpreter is denotational, *i.e.* is able to take any abstract property as input of a block and compute a sound over-approximation of the states reachable from it. In this work we have computed an invariant for each program point but Astrée spares some computations keeping as few invariants as possible during iteration, and we should also consider that approach.

The current paper only handles numeric abstractions. Astrée uses a lot of such abstractions but in order to analyse C programs finely those abstractions interact with an abstract memory domain [Min06a] that we do not at all formalise here.

Building a realistic verified analyser would be helpful to test the scalability of the technique we provide in Section 2.2.2. The extracted code has been tested on few small programs but it remains to perform more large scale experiments.

Chapter 3

Toward Verified Static Analysis Tools for Java Bytecode Programs

My PhD work was mainly focused on methodology. Several small scale static analysers have been formalised for various fragments of the Java bytecode langage but I did not try to build a fully realistic analyser. Scaling was an objective, but more in term of efficiency of the generated code. For that purpose, a lattice library has been developed. It relies on efficient data structures as binary search trees and provides reusable difficult proofs for programming a static analyser and proving it terminates. What I did not experiment is how a mechanised soundness proof of a static analysis scales to a realistic programming language. This chapter reports on two attempts in this direction. Section 3.1 reports on the formalisation of an information flow verifier for a large fragment of the Java bytecode language. We explain why the proof was challenging when taking into account the unstructured nature of bytecode programs because of jumps and exceptions. Section 3.2 presents a verified data-race-free verifier for an other fragment of the Java bytecode. This time, we tackle concurrency features of the Java language and formalise a key component for a Java software toolchain. We focus the presentation on the various semantic layers we have introduced in the formalisation because we believe they provide a useful methodology for further developments. Section 3.4 gives conclusions about these works and motivates our goal for the next chapter.

3.1 Information Flow Analysis

Non-interference is a semantical condition on programs that guarantees the absence of illicit information flow throughout their execution. Starting from the work of Volpano and Smith [VS97], type systems have become a popular means to enforce information flow policies in programming languages [SM03]. Much of previous work on type systems for non-interference has focused on calculi or high-level programming languages, and existing type systems for low-level languages typically omit objects, exceptions, and method calls, and/or do not prove formally the soundness of the type system. We report in this section on an information flow type system for a sequential JVM-like language that includes classes, objects, arrays, exceptions and method calls. We have proved with the Coq proof assistant that the type system guarantees non-interference. We have extracted from our development a certified lightweight bytecode verifier for information flow. This work provides, to our best knowledge, the first sound and implemented information flow type system for such an expressive fragment of the JVM.

The technical details of this work are not in the scope of this document. The interested reader can consult a journal version for them [BPR12]. We will concentrate here on what we believe makes this work specially challenging and original.

3.1.1 The Non-interference Property for Java Programs

Program non-interference is generally based on a partition of the set of program variables into two disjoint sets: the high (or secret) variables and the low (or public) variables. A program is then said *non interferent* if the final values of the low variables after an execution do not depend on the initial values of the high variables. In order to express more formally this property, we have to introduce an indistinguishability relation $\sim$ on program states such that two states s_1, s_2 are indistinguishable (noted $s_1 \sim s_2$) if and only if they appear equal from the (partial) point of view of the attacker. In the simplest setting, it is sufficient to express indistinguishability by saying that all low variables have the same value both in s_1 and s_2. Suppose the execution of a program P is modelled with a function $[\![P]\!]$ that takes an input state s as argument and returns the final state obtained after the execution of P on s. P is said non interferent if and only iff

$$\forall s_1, s_2,\ s_1 \sim s_2 \implies [\![P]\!]s_1 \sim [\![P]\!]s_2$$

In our work, the set of variables is indeed the Java memory of a program with dynamically allocated objects and arrays, method parameters and local variables, and operand stack. It makes the definition of non-interference more technical. Each part of the Java memory must be labelled with visibility annotations. Moreover, the memory space grows during execution since objects are allocated. Therefore indistinguishability is defined relative to a bijection β on (a partial set of) locations in the Java heap. The bijection maps low objects (low objects are objects whose references might be stored in low fields or low variables) allocated in the heap of the first state s_1, to low objects allocated in the heap of the second state s_2. The objects might be indistinguishable, even if their locations are different during execution because allocation of low objects may be interleaved with allocation of high objects. The non-interference property now looks like the following:

$$\forall s_1, s_2, \forall \beta,\ s_1 \sim_\beta s_2 \implies \exists \beta', \beta \subseteq \beta' \wedge [\![P]\!]s_1 \sim_{\beta'} [\![P]\!]s_2$$

It reads as follow: if two initial states are indistinguishable with respect to a bijection β that maps the low objects of s_1 to the low objects of s_2, then after executing the program P, the bijection β can be extended to a bijection β' on a larger set of low objects such that the resulting final states are still indistinguishable. Such a bijection on heap domain has been introduced by Banerjee and Naumann [BN05].

3.1.2 Information Flow Type System for Unstructured Programs

In a high level language with structured control flow, typing judgements are of the form $\vdash c :\ k$ with c a command and k a security level (H for high or L for low). Informally, if a command c is typable then it is non-interfering, and moreover if $\vdash c :\ H$ then c does not modify any low variable. In such systems, indirect flows are prevented by the typing rules branching statements, which for if-then-else statements is usually of the form [VS97] (where by definition of $\leq$, $L \leq H$):

$$\frac{\vdash e :\ k \qquad \vdash c_1 :\ k_1 \qquad \vdash c_2 :\ k_2 \qquad k \leq k_1, k_2}{\vdash \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 :\ k}$$

and thus ensures that the write effects of c_1 and c_2 are not less or equal to the guard of the branching statement.

To prevent illicit flows in a low-level language with unstructured control flow, the typing rules for branching instructions cannot simply enforce local constraints, i.e. they cannot talk only about the current program point and its successors. Instead, the typing rules must also enforce global constraints that prevent low assignments and updates to occur under high guards. Therefore, the typing rules rely on a graph representation of the program, and an approximation of the scope of branching statements using *control dependence regions* (CDR).

Our type system assumes that programs are bundled with additional information about their control dependence regions. This assumption is in line with the intended usage of our type checker as a lightweight bytecode verifier and streamlines the presentation by allowing us to focus on the information flow analysis itself. The information is given in the form of two functions region and jun. The intuition behind regions and junction points is that $\mathsf{region}(i)$ includes all program points executing under the guard at i and that $\mathsf{jun}(i)$, if it exists, is the sole exit from the region of i; in particular, whenever $\mathsf{jun}(i)$ is defined there should be no return instruction in $\mathsf{region}(i)$. Figure 3.1 provides examples of regions of two compiled programs.

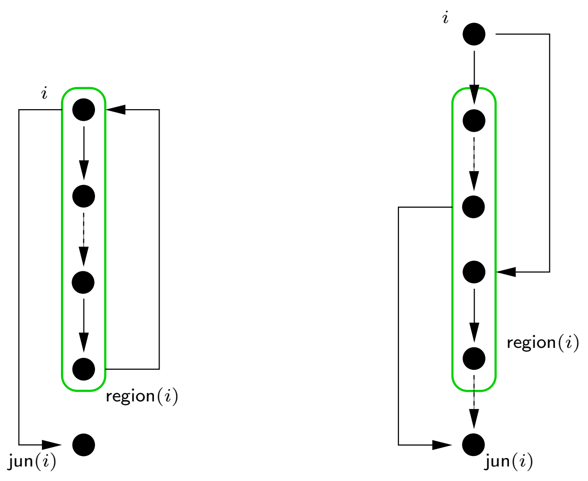


Figure 3.1: Example of CDR for a while (left part) and an if construct (right part).

The soundness of the type system requires that the functions verify the following properties: any successor of i either belongs to the region of i, or is equal to $\mathsf{jun}(i)$ (if defined), and $\mathsf{jun}(i)$ is the sole exit to the region of i; in particular if $\mathsf{jun}(i)$ is defined there should be no return instruction in $\mathsf{region}(i)$. When dealing with exceptions, we consider a family of region that are parameterised by the nature of the initial branching point (conditional or exception of a certain kind).

These properties are checked by a verified CDR checker. They can be computed efficiently with a post-dominator computation that does not need to be formally verified.

The type system is further parametrised by a security environment that attaches a security level to each program point. Informally, the security level of a program point is an upper bound of all the guards under which the program point executes. The security environment is used in conjunction with the CDR information to prevent implicit flows. This is done in two steps: on the one hand, the typing rule for branching statements enforces that the security environment of a program point is indeed an upper bound of the guard under which it executes; for instance, the rule for the conditional jump ifeq bytecode is of the form:

$$\frac{P[i] = \mathsf{ifeq}\ j \qquad \forall j' \in \mathsf{region}(i),\ k \leq se(j')}{i \vdash k :: st \Rightarrow \cdots}$$

Here j denotes the successor node of the instruction if the top of the operand stack is equal to zero. k is the security level attached to the top of the operand stack. We require every node j' which is control dependent with the node i to have a security environment $se(j')$ greater or equal than k.

On the other hand, the typing rules for instructions with write effect must check that the security level of the variable or field to be written is at least as high as the current security environment. For instance, the rule for the store of a local variable with the content of the first element of the operand stack (instruction store) becomes:

$$\frac{P[i] = \mathsf{store}\ x \qquad k \sqcup se(i) \leq \vec{k_v}(x)}{i \vdash k :: st \Rightarrow st}$$

Here k is the security level of the top of the operand stack. $\vec{k_v}(x)$ is the flow-insensitive security level attached to the local variable x. We require it to be greater or equal than both k and the current security environment $se(i)$.

The combination of both rules allows to prevent indirect flows. For instance, the standard example of indirect flow `if` $(\mathtt{y_H})$ $\{\mathtt{x_L} = 0;\}$ `else` $\{\mathtt{x_L} = 1;\}$ is compiled in a simplified bytecode language as

0 :	load y_H	*push the content of the high variable on the stack*
1 :	ifeq 5	*jump to 5 if the top of the stack is equal to zero, next line otherwise*
2 :	push 0	*push the constant 0 on the stack*
3 :	store x_L	*store the top of stack in the variable x*
4 :	goto 7	*jump to 7*
5 :	push 1	*push the constant 1 on the stack*
6 :	store x_L	*store the top of stack in the variable x*
7 :	...	

By requiring that $se(i) \leq \vec{k_v}(x)$, where i is the program point of the store instruction, and by requiring a global constraint on the security environment in the ifeq rule, the type system ensures that the above program will be rejected: $se(3)$ must be H because the store instruction is under the influence of a high ifeq at line 1 and $3 \in \mathsf{region}(1)$. Thus the transition for the store instruction cannot be typed.

3.1.3 Exceptional Control Flow

Exceptions introduce several potential sources of information leakage; in particular, attackers may infer sensitive information from the termination mode of programs. The method security signatures we consider follow the design proposed by Myers in the Jif tool [Mye99]. Method signatures are of the form

$$\vec{k_v} \xrightarrow{k_h} \vec{k_r}$$

The vector $\vec{k_v}$ provides the flow insensitive security level of all method variables (being a parameter or not). The *heap effect level* k_h is needed to make a modular analysis. It represents a lower bound for security levels of fields that are affected during the execution of the method. The security level $\vec{k_r}$ (called *output level*) is a list of security levels of the form $\{\mathsf{Norm} : k, e_1 : k_1, \dots e_n : k_n\}$, where k is the security level of the return value and e_i is an exception class that might be propagated by the method in a security environment (or due to an exception-throwing instruction) of level k_i. Therefore each exceptional path can be given a different security level.

The following Java method presents an example of a typable method.

```
int m(boolean x,C y) throws C {
  if (x) {throw new C();} else {y.f = 3;};
  return 1;
}
```

The method `m` may throw two kinds of exceptions: an exception of class `C` depending on the value of `x`, and an exception of class **np** depending on the values of `x` and `y`. Normal return depends on `y` because execution terminates normally only if it is not *null*. The method `m` is typable in our type system with the policy $m : (\mathit{this} : L,\ x : L,\ y : H) \xrightarrow{H} \{\mathsf{Norm} : H,\ C : L,\ \mathbf{np} : H\}$. As a consequence the caller of such a method may want to catch null-pointer exceptions after a call to `m` in order to prevent an information leak trough an abrupt termination by an uncaught null-pointer exception.

3.1.4 Formal Development

We have formalized within the Coq proof assistant the information flow type system for a sequential JVM-like language that includes classes, objects, arrays, exceptions and virtual method calls. Moreover, we have formalized executable checkers for the CDR properties and for typability, and proved formally their soundness—in the sense that an annotated program that is accepted by the CDR checker satisfies the CDR properties, and that an annotated program that is accepted by the information flow checker is typable w.r.t. our type system and non-interferent. The statement of soundness of the type system hinges on a formalization of the operational semantics of the JVM, and of the notion of non-interferent program.

The development relies on a formal semantics of the JVM in Coq, called Bicolano, which has been developed within the Mobius project to serve as a common basis for certification of proof carrying code technologies in Coq. Bicolano closely follows the official JVM specification—although some features are omitted, e.g. initialization, subroutines, multi-threading, dynamic class loading, garbage collection, 64-bit arithmetic and floats.

$$\frac{\begin{array}{c} P_m[pc] = \mathsf{invokevirtual}\ m_{\mathrm{ID}} \qquad m' = \mathsf{lookup}_P(m_{\mathrm{ID}}, \mathsf{class}(h(l))) \\ l \in dom(h) \qquad \mathsf{length}(os_1) = \mathsf{nbArguments}(m_{\mathrm{ID}}) \\ f' = [m', 1, \{\mathit{this} \mapsto l, \vec{x} \mapsto os_1\}, \varepsilon] \qquad f'' = [m, pc, \rho, os_2] \end{array}}{\langle h, [m, pc, \rho, l :: os_1 :: os_2], sf\rangle \to \langle h, f', f'' :: sf\rangle}$$

$$\frac{P_m[pc] = \mathsf{return}}{\langle h, [m, pc, \rho, v :: os], [m', pc', \rho', os'] :: sf\rangle \to \langle h, [m', pc' + 1, \rho', v :: os'], sf\rangle}$$

Figure 3.2: Small-step semantics rule for virtual method call and return

The core of Bicolano is a small-step operational semantics which describes the dynamic behavior of a bytecode program according to the JVM specification. The small-step semantics

$$\frac{\begin{array}{c}P_m[i] = \mathsf{invokevirtual}\ m_{\mathrm{ID}} \qquad m' = \mathsf{lookup}_P(m_{\mathrm{ID}}, \mathsf{class}(h(l)))\\ l \in \mathsf{dom}(h) \qquad \mathsf{length}(os_1) = \mathsf{nbArguments}(m_{\mathrm{ID}})\\ \langle 1, \{this \mapsto l, \vec{x} \mapsto os_1\}, \epsilon, h\rangle \leadsto^{+}_{m'} v, h'\end{array}}{\langle i, \rho, os_1 :: l :: os_2, h\rangle \leadsto_m \langle i+1, \rho, v :: os_2, h'\rangle}$$

Figure 3.3: Mix-step semantics rule for virtual method call

is formalized as an inductively defined relation $\cdot \to \cdot$ between states of the virtual machine, where a state consists of a heap, and a stack frame; Figure 3.2 presents the small-step semantics for method calls and return. In addition, Bicolano formalizes a mix-step semantics, in which method calls are performed in one step; in particular, the mix-step semantics for virtual method invocation appears in Figure 3.3. The mix-step semantics of a method m is also formalized as an inductively defined relation $\cdot \leadsto_m \cdot$ between states of the virtual machine, but uses a simplified notion of state in which the stack frame is replaced by a single frame. We do not detail these rules here. What really needs to be noticed here is the use of the transitive closure of $\leadsto_{m'}$ to directly obtain the result of the callee in the mix-step rule, while the small-step rule builds a new frame, pushes it on the top of the call stack and jumps to the first instruction of m'. Both semantics are equivalent, in the sense that the big-step semantics induced by the two semantics coincide; Bicolano formally establishes this equivalence between them. The crux of the proof is a lemma stating that each execution of the JVM to a final value implies the corresponding judgment of the mix-step semantics.

$$\left(\begin{array}{c}\langle h, [m, pc, \rho, os], \varepsilon\rangle \to^{*} \langle h', [m, pc', \rho', v' :: os'], \varepsilon\rangle\\ \text{with } P_m[pc'] = \mathsf{return}\end{array}\right) \implies \langle h, pc, l, s\rangle \leadsto^{+}_{m} (h', v')$$

A similar lemma is necessary for execution terminating with an uncaught exception.

We briefly comment on the role of the two semantics in our work: the mix-step semantics brings important simplifications in the definition of state indistinguishability and in the soundness proofs, and hence we use it to machine-check type soundness. The small-step semantics serves as a reference formalization, and hence the final theorem is stated using the small-step semantics.

3.1.5 A Typable Example Program

The CDR checker and the type checker are executable Coq programs that have been successfully extracted into Ocaml. We conclude this presentation of our information flow type checker with an example program on which we have tested these checkers. This *Tax Calculation* Java program is inspired from the case study proposed by Deng and Smith [DS04]. The program computes income taxes from an input array of taxable incomes and marital status. The program takes as argument an array `input` of inputs and a tax table `taxTable`. Then, for each index `i` in the array range, it performs a binary search to find an index `lo` such that

$$\mathtt{taxTable[lo].brackets} \leq \mathtt{input[i].taxableIncome} < \mathtt{taxTable[lo+1].brackets}$$

and updates the output array `out.tax[i]` with the computed tax (`taxTable[lo].married` or `taxTable[lo].single` depending on the marital status) and increment `out.married_nb` or `out.single_nb` to count the whole number of married and single tax returns. The taxable

```
class Output {
    int{L} single_nb;
    int{L} married_nb;
    int[]{L[H]} tax;

    Output{L}(int nbPeople) {
        single_nb = 0;
        married_nb = 0;
        tax = new int[nbPeople];
    }
    void updateMarried{L}(int{L} i, int{H} tax_data) {
        tax[i] = tax_data;
        married_nb++;
    }
    void updateSingle{L}(int{L} i, int{H} tax_data) {
        tax[i] = tax_data;
        single_nb++;
    }}

class Input {
    int{H} taxableIncome;
    boolean{L} maritalStatus;
}

class Tax {
    int{L} single;
    int{L} married;
    int{L} brackets;
}

class TaxCalculation {

    Output{L} main{L}(Input[]{L[L]} input, Tax[]{L[L]} taxTable) {
        Output{L} out = new Output(input.length);
        for (int{L} i=0; i < input.length; i++) {
            int{H} lo = 0;
            int{H} hi = taxTable.length;
            try { while (lo+1 < hi) {
                    int{H} mid = (lo + hi) / 2;
                    if (input[i].taxableIncome < taxTable[mid].brackets)
                      {hi = mid;} else {lo = mid;};
                  };
            } catch (NullPointerException e){};
            if (input[i].maritalStatus)
                { out.updateMarried(i,taxTable[lo].married);}
            else
                { out.updateSingle(i,taxTable[lo].single);};
        };
        return out;
    }
}
```

Figure 3.4: The Tax Calculation program.

incomes (field `taxableIncome`) and the array content of the income taxes (field `tax`) are given a high security level while other data are low.

Our type system allows a fine grain annotation on arrays. Array levels are of the form $k[k_c]$. These levels represent the security level of an array, distinguishing the level k_c of the content of the array (which could be itself an array) and the level k of the length of the array and of the reference itself. Hence an array of type $L[H]$ can be only updated in a high context (its content is high) but allocated in any context (its length and the value of its reference are low).

This program has been successfully type checked by our verified type checker and is therefore provably non-interferent. Even if the output of the binary search may releave confidential information, it does not affect the public information about the number of single and married tax returns. This property does not hold anymore if the programmer forget to put a `try ... catch (NullPointerException e) ...` around the search loop. Indeed if an attacker has accessed to the input `taxTable` array, he may force a null-pointer exception to be thrown when reaching a given range of taxable income during the search and then learn that the `input` array contains some taxpayers with high incomes.

3.2 Data Race Analysis

The second large formal development we report concerns *data races*, i.e., the situation where two threads access a memory location, and at least one of them changes its value, without proper synchronisation. This stands as a fundamental issue in multithreaded programming because it has direct impact on the semantics of a Java program. We will come back later to this point in Chapter 6. From now, we will say that a program is *data-race free* if no race occurs in any execution of this program. An execution is described as an *interleaving* of the actions performed by its threads. We study in this chapter a state-of-the-art race detection analysis proposed by Naik and Aiken [NAW06, NA07, Nai08] and formalize is soundness proof on a bytecode language.

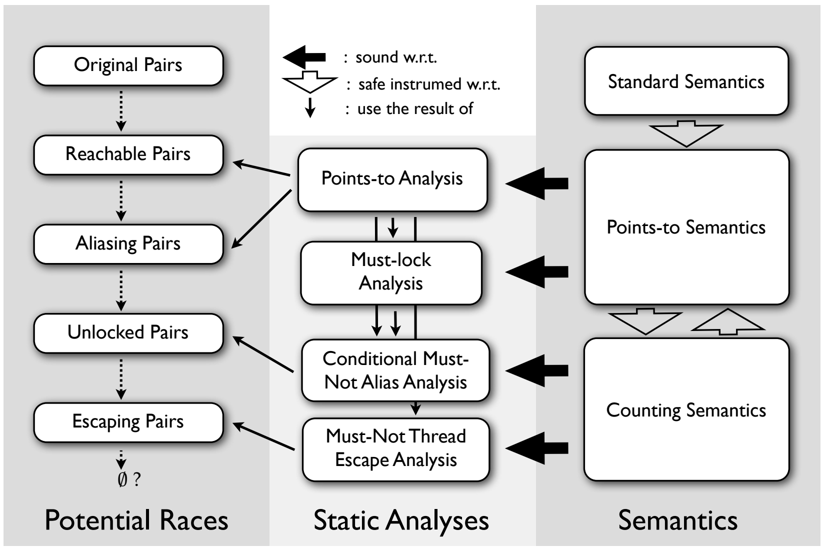


Figure 3.5: Architecture of the development

The general architecture of our development is sketched in Figure 3.5. We formalise four static analyses : a context-sensitive points-to analysis, a must-lock analysis, a conditional must-not alias analysis based on disjoint reachability and a must-not thread escape analysis. In order to ensure the data-race freeness of the program, these analyses are used to refine, in several stages an initial over-approximation of the set of potential races of a program, with the objective to obtain an empty set at the very last stage. Each analysis is mechanically proved correct with respect to an operational semantics. However, we consider three variants of semantics. While the first one is a standard small-step semantics, the second one attaches context information to each reference and frame. This instrumentation makes the soundness proof of the points-to analysis easier. The last semantics handles more instrumentation in order to count method calls and loop iterations. Each instrumentation is proved correct with respect to the semantics just above it. The notion of safe instrumentation is formalised through a standard simulation diagram.

3.2.1 A Challenging Example Program

```
class Main() {
    void main() {
        List l = null;
        while (*) {
            List temp = new List;
         1: temp.val = new T;
         2: temp.val.f = new A;
         3: temp.next = l;
            l = temp;
        };
        while (*) {
            T t = new T;
         4: t.data = l;
            t.start();
         5: t.f = null;
        }
    }
}
```

```
class A{};

class List{  T val;  List next; };

class T extends java.lang.Thread {
    A f;
    List data;
    void run(){
        while(*) {
         6: List m = this.data;
         7: while (*) { m = m.next; }
            synchronized(m)
         8:     { m.val.f = new A; }};
        }
    }
}
```

Figure 3.6: A challenging example program

Figure 3.6 presents an example of a source program for which it is challenging to formally prove race freeness. This example is adapted from the running example given by Naik and Aiken [NA07] in a While language syntax. The program starts in method `main` by creating in a first loop, a simple linked list `l` and then launches a bunch of threads of class `T` that all share the list `l` in their field `data`. Each thread, launched in this way, chooses non deterministically a cell `m` of the list and then updates `m.val.f`, using a lock on `m`.

Figure 3.7 presents the potential races computed for this example. A data race is described by a triplet (i, f, j) where i and j denote the program points in conflicts and f denotes the accessed field. The first over-approximation, the set of *Original Pairs* is simply obtained by typing: thanks to Java's strong typing, a pair of accesses may be involved in a race only if both access the same field and at least one access is a write. For each other approximation, a marked indicates a potential race. The program is in fact data race free but the size of the set of *Original Pairs* (13 pairs here) illustrates the difficulty of statically demonstrating it.

Following [NAW06], a first approximation of races computes the field accesses that are

Original	Reachable	Aliasing	Unlocked	Escaping
$(1,\texttt{val},1),(1,\texttt{val},2),(2,\texttt{f},2),$ $(3,\texttt{next},3),(4,\texttt{data},4)$		✓	✓	
$(5,\texttt{f},5)$		✓	✓	✓
$(2,\texttt{f},5)$			✓	
$(5,\texttt{f},8)$	✓		✓	✓
$(4,\texttt{data},6),(3,\texttt{next},7),(1,\texttt{val},8),$ $(2,\texttt{f},8)$	✓	✓	✓	
$(8,\texttt{f},8)$	✓	✓		✓

Figure 3.7: Potential race pairs in the example program

reachable from the entry point of the program and removes also pairs where both accesses are taken by the main thread (*Reachable pairs*). Some triplets may also be removed with an alias analysis that shows that two potential conflicting accesses are not in alias (*Aliasing pairs*). Both sets rely on a points-to analysis. Among the remaining potential races, several triplets can be disabled by a *must-not thread escape* analysis that predicts a memory access only concerns a reference which is local to a thread at the current point (*Escaping pairs*). The last potential race $(8,\texttt{f},8)$ requires the most attention since several threads of class `T` are updating fields `f` in parallel. These writes are safe because they are guarded by a synchronization on an object which is the only ancestor of the write target in the heap. Such reasoning relies on the fact that if locks guarding two accesses are different then so are the targeted memory locations. This last triplet is removed by a *conditional must not alias.*

In this presentation, we will focus on the methodological contribution of this work that, we believe, relies in the various semantic layers we have introduced. The details of the formalised static analysis will stay at an informal level. More details can be found in [DP09].

3.2.2 Standard Semantics

The bytecode language we consider in this formalisation is a subset of the Java bytecode language that is expressive enough to represent the previous example. The instructions set allows to manipulate objects, call virtual methods, start threads and lock (or unlock) objects for threads synchronization. Compared to real Java, we discard all numerical manipulations because they are not relevant to our purpose. Static fields, static methods and arrays are not managed here but they are nevertheless source of several potential data races in Java programs. Naik's approach [NAW06] for these layers is similar to the technique developed for objects. At last, as Naik and Aiken did before us, we only cover synchronization by locks without join, wait and interruption mechanisms. Our approach is sound in presence of such statements, but doesn't take into account the potential races they could prevent. The last missing feature is the Java's exception mechanism. Exceptions complicate the control flow of a Java program. We expect that handling this mechanism would increase the amount of formal proof but will not require new enlightening proof techniques.

Even under these restriction, the semantic domain of our language is already large. States are defined in Figure 3.8, where $\to$ stands for total functions and $\rightharpoonup$ stands for partial functions. We fix a set $\mathbb{C}_{id}$ of classes, a set $\mathbb{F}$ of fields, a set $\mathbb{V}$ of local variables and a set $\mathbb{M}$ of method. We distinguish *location* ($\mathbb{L}$) and *memory location* ($\mathbb{O}$) sets. The set of locations is kept abstract in this presentation. In this section, a memory location is itself a location ($\mathbb{L} = \mathbb{O}$). This

$$\begin{array}{rclll}
\mathbb{L} & \ni & \ell & & \text{(location)} \\
\mathbb{O} = \mathbb{L} & \ni & \ell & & \text{(memory location)} \\
\mathbb{O}_\perp & \ni & v & ::= \ \ell \mid \texttt{Null} & \text{(value)} \\
 & & s & ::= \ v :: s \mid \varepsilon & \text{(operand stack)} \\
\mathbb{V} \to \mathbb{O}_\perp & \ni & \rho & & \text{(local variables)} \\
\mathbb{O} \rightharpoonup \mathbb{C}_{id} \times (\mathbb{F} \to \mathbb{O}_\perp) & \ni & \sigma & & \text{(heap)} \\
PPT = \mathbb{M} \times \mathbb{N} & \ni & ppt & ::= \ (m, i) & \text{(program point)} \\
CS & \ni & cs & ::= \ (m, i, s, \rho) :: cs \mid \varepsilon & \text{(call stack)} \\
\mathbb{O} \rightharpoonup CS & \ni & L & & \text{(thread call stacks)} \\
\mathbb{O} \to ((\mathbb{O} \times \mathbb{N}^*) \cup \{\texttt{free}\}) & \ni & \mu & & \text{(locking state)} \\
 & & st & ::= \ (L, \sigma, \mu) & \text{(state)} \\
 & & e & ::= \ \tau \mid (\ell, ?_f^{ppt}, \ell') \mid (\ell, !_f^{ppt}, \ell') & \text{(event)}
\end{array}$$

Figure 3.8: States and actions

redundancy will be useful when defining new instrumented semantics where memory locations will carry more information in the next section. In a state (L, σ, μ), L maps memory locations (that identify threads) to call stacks, σ denotes the heap that associates memory locations to objects (c, map) with c a class and map a map from fields to values. A locking state μ associates with every location ℓ a pair (ℓ', n) if ℓ is locked n times by ℓ' and the constant `free` if ℓ is not held by any thread (Java locks are re-entrant).

The dynamic semantics of our language is defined over states as a labelled transition system. Labels, or events, are also defined in Figure 3.8. Labelled transitions have the form $st \xrightarrow{e} st'$. To keep the presentation short, we do not detail the definition of this relation and refer the interested reader to [DP09]. An event $(\ell, ?_f^{ppt}, \ell')$ (resp. $(\ell, !_f^{ppt}, \ell')$) denotes a read (resp. a write) of a field f, performed by the thread ℓ over the memory location ℓ', at a program point ppt. An event τ denotes a silent action.

We write $RState(P)$ for the set of states that contains the initial state of a program P, and that is closed by reduction. A data race is a tuple (ppt_1, f, ppt_2) such that $Race(P, ppt_1, f, ppt_2)$ holds.

$$\frac{st \in RState(P) \quad st \xrightarrow{\ell_1 !_f^{ppt_1} \ell_0} st_1 \quad st \xrightarrow{\ell_2 \mathcal{R} \ell_0} st_2 \quad \mathcal{R} \in \{?_f^{ppt_2}, !_f^{ppt_2}\} \quad \ell_1 \neq \ell_2}{Race(P, ppt_1, f, ppt_2)}$$

l_1 and l_2 are two distinct locations on which memory actions can be perfomed from the current state st and l_1's action is a write.

The ultimate goal of our certified analyser is to guarantee *Data Race Freeness*, *i.e.* for all $ppt_1, ppt_2 \in PPT$ and $f \in \mathbb{F}$, $\neg Race(P, ppt_1, f, ppt_2)$.

3.2.3 Points-to Semantics

Naik and Aiken make intensive use of points-to analysis in their work. Points-to analysis computes a finite abstraction of the memory where locations are abstracted by their allocation site. The analysis can be made context sensitive if allocation sites are distinguished wrt. the calling context of the method where the allocation occurs.

```
Module Type CONTEXT.

    Parameter pcontext : Set. (* pointer context *)
    Parameter mcontext : Set. (* 𝕄 context *)

    Parameter make_new_context  : 𝕄 -> ℕ -> ℂ_id  -> mcontext -> pcontext.
    Parameter make_call_context  : 𝕄 -> ℕ -> mcontext -> pcontext -> mcontext.
    Parameter get_class : program -> pcontext -> option ℂ_id .

    Parameter class_make_new_context :  ∀  p m i cid c,
      body m i = Some (New cid) -> get_class p (make_new_context m i cid c) = Some cid.

    Parameter init_mcontext : mcontext.
    Parameter init_pcontext : pcontext.

    Parameter eq_pcontext :  ∀  c1 c2:pcontext, {c1=c2}+{c1<>c2}. // executable equality test
    Parameter eq_mcontext :  ∀  c1 c2:mcontext, {c1=c2}+{c1<>c2}. // executable equality test

End CONTEXT.
```

Figure 3.9: The Module Type of Points-to Contexts

Many static analyses use this kind of information to have a conservative approximation of the call graph and the heap of a program. Such analyses implicitly reason on instrumented semantics that directly manipulates informations on allocation sites while a standard semantics only keeps track of the class given to a reference during its allocation. In this section we formalise such an intermediate semantics.

This *points-to semantics* takes the form of a Coq module functor

```
Module PointsToSem (C:CONTEXT). ... End PointsToSem.
```

parameterised by an abstract notion of context which captures a large variety of points-to contexts. Figure 3.9 presents this notion. A context is given by two abstract types `pcontext` and `mcontext` for pointer contexts and method contexts. Function `make_new_context` is used to create a new pointer context `(make_new_context m i cid c)` when an allocation of an object of class `cid` is performed at line `i` of a method `m`, called in a context `c`. We create a new method context `(make_call_context m i c p)` when building the calling context of a method called on an object of context `p`, at line `i` of a method `m`, itself called in a context `c`. At last, `(get_class prog p)` allows to retrieve the class given to an object allocated in a context `p`. The hypothesis `class_make_new_context` ensures consistency between `get_class` and `make_new_context`.

The simplest instantiation of this semantics takes class name as pointer contexts and uses a singleton type for method context. A more interesting instantiation is k-objects sensitivity: contexts are sequences of at most k allocation sites $(m, i) \in \mathbb{M} \times \mathbb{N}$. When creating an object at site (m, i) in a context c, we attach to this object a pointer context $(m, i) \oplus_k c$ defined by $(m, i) \cdot c'$ if $|c|{=}k$, $c = c' \cdot (m', i')$ and $(m, i) \cdot c$ if $|c| < k$, without any change to the current method context. When calling a method on an object we build a new frame with the same method context as the pointer context of the object.

The definition of this semantics is similar to the standard semantics described in the previous section, except that memory locations are now couples of the form (ℓ, p) with ℓ a location and p a pointer context. We found convenient for our formalisation to deeply instrument the heap that is now a partial function from memory location of the form (ℓ, p) to objects. This allows us to state a property about the context of a location without mentioning the current heap (in contrast to the class of a location in the previous standard semantics). The second change concerns frames that are now of the form (m, i, c, s, ρ) with c being the method context of the current frame.

In order to reason on this semantics and its different instantiations we give to this module a module type `POINTSTO_SEM` such that for all modules `C` of type `CONTEXT`, `PointsToSem(C) : POINTSTO_SEM`.

Several invariants are proved on this semantics, for example that if any memory locations (ℓ, p_1) and (ℓ, p_2) are in the domain of a heap reachable from a initial state, then $p_1 = p_2$.

```
Module PointsToSemInv (S:POINTSTO_SEM). ...
    Lemma reachable_wf_heap : ∀ prog L sigma mu,
        reachable prog (L,sigma,mu) →
        ∀ l p1 p2, sigma (l,p1)<>None → sigma (l,p2)<>None → p1=p2.
    Proof. ... Qed.
End PointsToSemInv.
```

Safe Instrumentation The analyses we formalise on top of this points-to semantics are meant for proving absence of race. To transfer such a semantic statement in terms of the standard semantics, we prove simulation diagrams between the transitions systems of the standard and the points-to semantics. Such diagram then allows us to prove that each standard race corresponds to a points-to race.

```
Module SemEquivProp (S:POINTSTO_SEM).
    Lemma race_equiv :
      ∀ p ppt ppt', Standard.race p ppt ppt' → S.race p ppt ppt'.
    Proof. ... Qed.
End SemEquivProp.
```

Here `Standard.race` denotes the predicate *Race* that we have introduced in the previous section.

Points-to Analysis A generic context-sensitive analysis is specified as a set of constraints attached to a program. The analysis is flow-insensitive for heap and flow-sensitive for local variables and operand stacks. Its result is given by four functions

```
  PtL: mcontext -> 𝕄 -> ℕ -> 𝕍 -> (pcontext -> Prop).
  PtS: mcontext -> 𝕄 -> ℕ -> list (pcontext -> Prop).
  PtR: mcontext -> 𝕄 -> (pcontext -> Prop).
  PtF: pcontext -> 𝔽 -> (pcontext -> Prop).
```

that attach pointer context properties to local variables (`PtL`), operand stack (`PtS`) and method returns (`PtR`). `PtF` is the flow-insensitive abstraction of the heap.

This analysis is parameterized by a notion of context and proved correct with respect to a suitable points-to semantics. The final theorem says that if `(PtL,PtS,PtR,PtF)` is a solution of the constraints system then it correctly approximates any reachable states. The notion of correct approximation expresses, for example, that all memory locations (ℓ, p) in the local variables ρ or operand stack s of a reachable frame (m, i, c, s, ρ) is such that p is in the points-to set attached to `PtL` or `PtS` for the corresponding flow position (m, i, c).

This static analysis and all the analyses that we have formalised in this work are non-executable in Coq. Each of them is specified as a fixpoint problem using inductive definitions. When relying on set data structures, we simply consider predicates. We discuss further this design in the conclusion of the chapter.

Must-Lock Analysis Fine lock analysis requires to statically understand which locks are definitely held when a given program point is reached. For this purpose we specify and prove correct a flow sensitive must-lock analysis that computes at each flow position, an under-approximation of the local variables that are currently held by the thread reaching this position. The specification of `Locks` depends on the points-to information `PtL` computed before. This is a useful information for the bytecode `monitorexit` instruction because the unlocking of a variable x can only cancel the lock information of the variables that may be in alias with x.

Removing False Potential Races The previous points-to analysis supports the first two stages of the race analyser of Naik et al [NAW06]. The first stage prunes the so called *ReachablePairs*. It only keeps in *OriginalsPairs* the accesses that may be reachable from a `start()` call site that is itself reachable from the `main` method, according to the points-to information. Moreover, it discards pairs where each accesses are performed by the main thread because there is only one thread of this kind.

The next stage keeps only the so called *AliasingPairs* using the fact that a conflicting access can only occur on references that may alias. In the example of the Figure 3.6, the potential race $(5, \mathtt{f}, 8)$ is cancelled because the points-to information of `t` and `m.val` are disjoint.

For each stage we formally prove that all these sets over-approximate the set of real races wrt. the points-to semantics.

Extension to a Counting Semantics The last two stages of our analysis require a even deeper instrumentation for counting method calls and loop iterations. In addition to the allocation site (m, i) and the calling context c of an allocation, this semantics captures counting information. More precisely, it records that the allocation occurred after the n^{th} iteration of a flow edge (i, j) in the k^{th} call to m in context c. To record this information during allocation, the semantics counts, and records, iteration of all flow edges and all method call (in a given points-to context) in its instrumented state. We hence update the semantic domain as follows:

$$\begin{array}{rcllll}
mVect & = & \mathbb{M} \times \mathtt{mcontext} \rightarrow \mathbb{N} & \ni & \omega & \text{(method vector)} \\
lVect & = & \mathbb{M} \times \mathtt{mcontext} \times (\mathbb{N} \times \mathbb{N}) \rightarrow \mathbb{N} & \ni & \pi & \text{(iteration vector)} \\
 & & CP & \ni & cp ::= \langle m, i, c, \omega, \pi \rangle & \text{(code pointer)} \\
\mathbb{O} & = & \mathbb{L} \times CP & & & \text{(memory location)} \\
 & & CS & \ni & cs ::= (cp, s, \rho) :: cs \mid \varepsilon & \text{(call stack)} \\
 & & & & st ::= (L, \sigma, \mu, \omega_g) & \text{state}
\end{array}$$

A frame holding the code pointer $\langle m, i, c, \omega, \pi \rangle$ is the $\omega(m, c)^{th}$ call to method m in context c (a k-context) since the execution began and, so far, it has performed $\pi(m, c, \phi)$ steps through edge ϕ of its control flow graph. In a state $(L, \sigma, \mu, \omega_g)$, ω_g is a global method vector used as a shared call counter by all threads.

As we did between the standard and the points-to semantics we prove a diagram simulation between the points-to semantics and the counting semantics. It ensures that all *points-to races* correspond to a *counting race*. However, in order to use the soundness theorem of the must-lock analysis we also need to prove a bisimulation diagram. It ensures that all states that are reachable in the counting semantics correspond to a reachable state in the points-to semantics. It allows us to transfer the soundness result of the must-lock analysis in terms of the counting semantics.

We use this last semantic layers to formalise two last static analyses. The first is an original loop-sensitive escape analysis. It guarantees that, at some program point, the last object allocated by a thread at a given allocation site is still local to that thread. In particular, our analysis proves that an access performed at point 4 in our running example, is on the last object of type T allocated by the main thread (which is is local, although at each loop iteration, the new object eventually escapes the main thread). The second is based on the notion of *Conditional Must Not Alias Analysis* [NA07]. It provides an under-approximation DR_Σ of the notion of *disjoint reachability*. Given a set of heaps Σ and a set of allocation sites H, the disjoint reachability set $DR_\Sigma(H)$ is the set of allocation sites h such that whenever an object o allocated at site h may be reachable by one or more field dereferences for some heap in Σ, from objects o_1 and o_2 allocated at any sites in H then $o_1 = o_2$. It allows to remove the last potential race of our running example.

3.3 Related work

Information flow type soundness Jif [Mye99] is a state-of-the-art information-flow typed extension of Java that builds upon the decentralised label model to support flexible and expressive information flow policies. Jif offers developers a practical tool for ensuring that applications meet their information flow policies, but lacks a soundness proof. However, Banerjee and Naumann [BN05, Nau05] have shown the soundness of a simpler information flow type system for a fragment of Java with objects and methods. Our work extend their work taking into account exceptions, unstructured control flow and arrays.

Hammer and Snelting [HS09] have developed an information flow analysis based on control dependence regions; they use path conditions to achieve precision in their analysis, and to exhibit security leaks if the program is insecure. The related static analysis have been formalised by Wasserrab [Was10] in Isabelle.

More recently, Popescu et al. [PHN12] compare various definitions of non-interference in a concurrent setting. Their theorems have been mechanically verified in Isabelle/HOL.

Data races Most of the recent machine checked semantics proofs do not consider multi-threading. Indeed, several formalisation of the sequential JVM and its type system have been performed (notably the work of Klein and Nipkow [KN06]), but few have investigated its multithreaded extension. Hobor *et al.* [HAN08] define a modular operational semantics for Concurrent C minor and prove the soundness of a concurrent separation logic w.r.t. it, in Coq. Petri and Huisman [PH08] propose a realistic formalization of multithreaded Java bytecode in Coq, Bicolano MT that extends the sequential semantics Bicolano considered in Section 3.1. Aspinall and Sevcik formalise the Java *data race free guarantee* theorem that ensures that data races free program can only have sequentially consistent behaviors. The work of Petri and Huisman [HP07] follows a similar approach. Recently, Lochbihler [Loc12] extended Aspinall and Sevcik's formalization by including infinite executions and dynamic allocations and connecting the Java Memoty Model to Klein and Nipkow semantics. Lochbihler [Loc10] also formalises the translation phase from Java source code to Java bytecode. The only machine checked proof of a data race analyser we are aware of is the work of Lammich and Müller-Olm [LMO07]. Their formalisation is done at the level of an abtract semantics of a flowgraph-based program model. They formalise a locking analyses with an alias analysis technique simpler than the one used by Naik and Aiken. The data race analysis heavily rely on points-to analysis. As far as we know, our formalisation is the first mechanised result in this area. Recently Leroy and Robert [LR12] have developed a points-to analysis in the CompCert framework. They have been able to extract an executable analyser from their development while our analysis is non-executable. In contrast, we provide a more general points-to framework that can be instantiated with various context-sensitive analyses.

3.4 Conclusions and Perspectives

The two verification techniques that have been formalised here are admittedly complex. They deal with subtle semantic properties and their enforcement mechanisms are based on a fine collaboration between various verification components. Their presentation would have been easier if done on a simpler core language. But our primary goal was to be as close as possible to the details that need to be taken into account during the implementation of such techniques.

I believe these two examples show important examples of verified and realistic static analyses but the underlying proof effort has been too important in my opinion. Each proof was more than 15.000 lines of Coq long and took more than one man-year. The data race analysis took even more time and we did not reach a formalisation that is fully executable. In itself, implementing such a static analysis would probably requires one more year (without a good static analysis library). After this observation, it was clear to me that one need to build a suitable static analysis platform that will facilitate both the implementation and the proof of static analyser. My PhD students Laurent Hubert and Delphine Demange have helped me in approaching such a goal. I will explain how in the next two chapters.

Chapter 4

Foundations and Implementation of Static Analysis for Java Bytecode Programs

Proving the correctness of a realistic static analysis can be time consuming. We have learnt this lesson from the previous Chapter. Implementing a realistic static analysis can also be time consuming. People have learnt this lesson a long time ago. To reduce this effort, static analysis platforms have been developped [VRCG$^+$99, IBM, BCF$^+$99] for several years. They provide reusable libraries that facilitate the development.

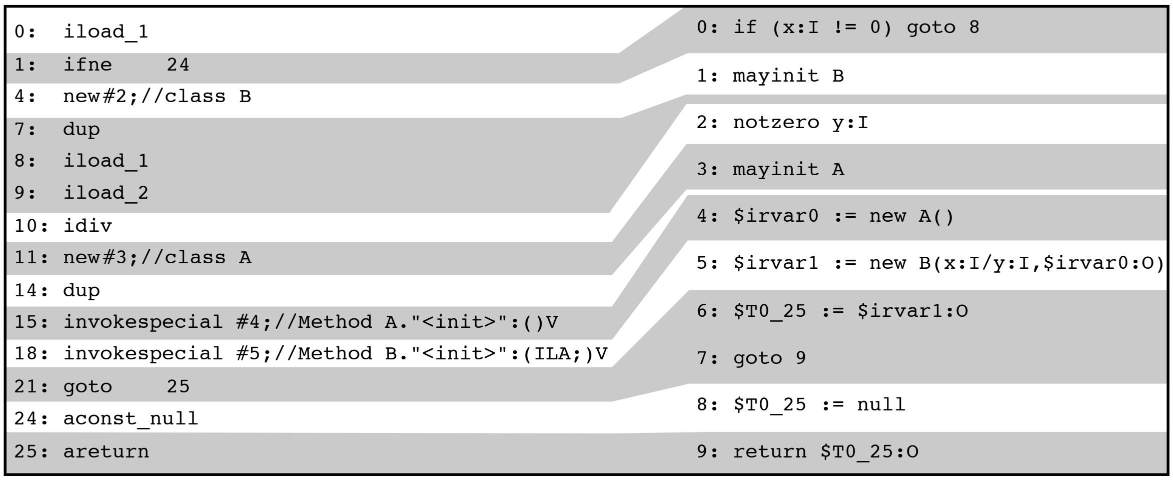
When Laurent Hubert started his PhD, our objective was to develop a rich OCaml library that would fulfil a similar task. To reach this goal, we have been studying several static analyses for Java [HJP08a, HP09, HJMP10, JKP11] from two angles: soundness proof and implementation. We quickly understood that a good intermediate representation was a key feature for such a platform and my second PhD student, Delphine Demange, specifically dealt with this task [DJP10]. This chapter gives a short presentation of the library we built (Section 4.1) and two examples of work we did, based on this platform, in Section 4.2 and 4.3.

In both case, we follow a same and original methodology. Each static analysis is evaluated in two complementary ways. First, its soundness is formalised in the Coq proof assistant, based on an idealised object-oriented language. Then, its precision is experimentally evaluated by building a complete tool that is run on a large number of programs. Even a sound static analysis or a sound type system is useless if it systematically returns "I don't know" or reject any interesting program. Validating their *precision* is hence an important objective.

4.1 Sawja and its Stackless Intermediate Representation

We present in this section the Sawja library, the first OCaml library providing state-of-the-art components for writing Java static analyzers in OCaml. The library represents an effort of 1.5 man-year and approximately 22,000 lines of OCaml (including comments) of which 4,500 are for the interfaces. Many design choices are based on our long-term goal of designing such a library using verified extracted code. The primary goal of the current version is compliance with the full Java language and efficiency of the data structures that underlies the program representation. The main features of Sawja are:

- parsing of `.class` files into OCaml structures and unparsing of those structures back into `.class` files;
- decompilation of the bytecode into a high-level stack-less IR;
- sharing of complex objects both for memory saving and efficiency purpose (structural equality becomes equivalent to pointer equality and indexation allows fast access to



```
0:  iload_1
1:  ifne     24
4:  new#2;//class B
7:  dup
8:  iload_1
9:  iload_2
10: idiv
11: new#3;//class A
14: dup
15: invokespecial #4;//Method A."<init>":()V
18: invokespecial #5;//Method B."<init>":(ILA;)V
21: goto     25
24: aconst_null
25: areturn
```

```
0: if (x:I != 0) goto 8
1: mayinit B
2: notzero y:I
3: mayinit A
4: $irvar0 := new A()
5: $irvar1 := new B(x:I/y:I,$irvar0:O)
6: $T0_25 := $irvar1:O
7: goto 9
8: $T0_25 := null
9: return $T0_25:O
```

Figure 4.1: Example of bytecode (left) and its corresponding IR (right) with colors to make explicit the boundaries of related code fragments

tables indexed by class, field or method signatures, etc.);

- the determination of the set of classes constituting a complete program (using several algorithms, including Rapid Type Analysis (RTA) [BS96]);
- a careful translation of many common definitions of the JVM specification, *e.g.*, about the class hierarchy, field and method resolution and look-up, and intra- and inter-procedural control flow graphs.

The library is developed by the Celtique research group and is freely available at

http://sawja.inria.fr/

We focus our presentation on the intermediate representation for which we have specially studied correctness. Section 4.5 will discuss correctness of the other main Sawja components.

The JVM is a stack-based virtual machine and the intensive use of the operand stack makes it difficult to adapt standard static analysis techniques that have been first designed for more classic variable-based codes. Hence, several bytecode optimization and analysis tools work on a bytecode *intermediate representation* (IR) that makes analyses simpler [BCF$^+$99, VRCG$^+$99]. Surprisingly, the semantic foundations of these transformations have received little attention. The transformation that is informally presented here has been formally studied and proved semantics-preserving in [DJP10]. Delphine Demange and myself have implemented it in the Sawja library together with an SSA variant that is not discussed here. The SSA representation is discussed in detail in Chapter 5.

Figure 4.1 gives the bytecode and IR versions of the simple method

```
B f(int x, int y) { return (x==0)?(new B(x/y, new A())):null;}
```

The bytecode version reads as follows: the value of the first argument `x` is pushed on the stack at program point 0. At point 1, depending on whether `x` is zero or not, the control flow jumps to point 4 or 24 (in which case the value `null` is returned). At point 4, a new object of class `B` is allocated in the heap and its reference is pushed on top of the operand stack. Its address is then duplicated on the stack at point 7. Note the object is *not initialised* yet.

Before the constructor of class `B` is called (at point 18), its arguments must be computed: lines 8 to 10 compute the division of `x` by `y`, lines 11 to 15 construct an object of class `A`. At point 18, the non-virtual method `B` is called, consuming the three top elements of the stack. The remaining reference of the `B` object is left on the top of the stack and represents from now on an *initialised* object.

The right side of Figure 4.1 illustrates the main features of the IR language. First, it is *stack-less* and manipulates *structured expressions*, where variables are annotated with *types*. For instance, at point 0, the branching instruction contains the expression $\texttt{x} : \texttt{I}$, where `I` denotes the type of Java integers. Another example of recovered structured expression is $\texttt{x} : \texttt{I}/\texttt{y} : \texttt{I}$ (at point 5). Second, expressions are *error-free* thanks to explicit checks: for instance, the instruction `notzero` $\texttt{y} : \texttt{I}$ at point 2 ensures that evaluating $\texttt{x} : \texttt{I}/\texttt{y} : \texttt{I}$ will not raise any error. Explicit checks additionally guarantee that the order in which *exceptions* are raised in the bytecode is preserved in the IR. Next, the object creation process is syntactically simpler in the IR because the two distinct phases of (i) allocation and (ii) constructor call are merged by *folding* them into a single IR instruction (see point 4). In order to simplify the design of static analyses on the IR, we forbid side effects in expressions. Hence, the nested object creation at source level is decomposed into two intermediate assignments (`$irvar0` and `$irvar1` are temporary variables introduced by the transformation). Notice that because of side-effect free expressions, the order in which the `A` and `B` objects are allocated must be reversed. Still, the IR code is able to preserve the *class static initialization order* using the dedicated instruction `mayinit` that calls the static class initializer whenever it is required.

The transformation algorithm is based on a symbolic execution of the bytecode, using a stack of symbolic expressions. The algorithm traverses the bytecode method instruction array, and for each instruction, given an input symbolic stack, produces some IR code (potentially a nop instruction), and modifies the symbolic stack according to the instruction being transformed. For the example given in Figure 4.1, the symbolic stack looks as follow.

$$\begin{array}{lr}
0: & \varepsilon \\
1: & \texttt{x} :: \varepsilon \\
4: & \varepsilon \\
7: & \texttt{B}_4 :: \varepsilon \\
8: & \texttt{B}_4 :: \texttt{B}_4 :: \varepsilon \\
9: & \texttt{x} :: \texttt{B}_4 :: \texttt{B}_4 :: \varepsilon \\
10: & \texttt{y} :: \texttt{x} :: \texttt{B}_4 :: \texttt{B}_4 :: \varepsilon \\
11: & \texttt{x/y} :: \texttt{B}_4 :: \texttt{B}_4 :: \varepsilon \\
14: & A_{11} :: \texttt{x/y} :: \texttt{B}_4 :: \texttt{B}_4 :: \varepsilon \\
15: & A_{11} :: A_{11} :: \texttt{x/y} :: \texttt{B}_4 :: \texttt{B}_4 :: \varepsilon \\
18: & \texttt{\$irvar0} :: \texttt{x/y} :: \texttt{B}_4 :: \texttt{B}_4 :: \varepsilon \\
21: & \texttt{\$irvar1} :: \varepsilon \\
24: & \varepsilon \\
25: & \texttt{null} :: \varepsilon
\end{array}$$

Before executing line 0 we start with an empty stack. After symbolic execution of `iload_1` the symbol `x` on top of the stack (`x` denotes is the first local variable while `y` denotes the second). For a i : `new` C instruction, we generate a special symbol C_i which identifies the line where the instruction is found. After an `invokespecial` instruction we substitute every occurrence of such a symbol by a fresh variable. For example $\texttt{B}_4$ is replaced by `$irvar1`. Such a symbolic execution is possible because the bytecode verifier impose restriction on their usage of operand stack. At each point of a program, every incoming paths must have an incoming operand stack of the same size.

The correctness theorem we have established for this transformation has to take into account the reordering of allocation that the transformation generates. Therefore it relies on an observation of heaps that are parameterised by bijection, in a similar manner than in Section 3.1. Its proof has not been machine-checked yet but rigorously proofread [DJP09].

The efficiency of the implementation has also been experimentally validated in [HBB+10]. Because the transformation processes in one pass, it performs more than 10 times faster than Soot [VRCG+99] which relies on iterative techniques, even when Soot does not try to optimize the code it produces (otherwise Soot is even slower). Furthermore, the number of local variables introduced by the intermediate representation is kept manageable and comparable to Soot when Soot uses other analyses to reduce their number.

4.2 Enforcing Secure Object Initialization in Java

The initialization of an information system is usually a critical phase where essential defense mechanisms are being installed and a coherent state is being set up. In object-oriented software, granting access to partially initialised objects is consequently a delicate operation that should be avoided or at least closely monitored. Indeed, the CERT recommendation for secure Java development [CER10a] clearly requires to *not allow partially initialised objects to be accessed* (guideline OBJ04-J). The CERT has assessed the risk if this recommendation is not followed and has considered the severity as *high* and the likelihood as *probable*. They consider this recommendation as a first priority on a scale of three levels.

The Java language and the Java Byte Code Verifier (BCV) enforce some properties on object initialization, *e.g.* about the order in which constructors of an object may be executed, but they do not directly enforce the CERT recommendation. Instead, Oracle provides a guideline that enforces the recommendation. Conversely, failing to apply this guidelines may silently lead to security breaches. In fact, a famous attack [DFW96] used a partially initialised class loader for privilege elevation.

We propose a twofold solution: (i) a modular type system which allows to express the initialization policy of a library or program, *i.e.* which methods may access partially initialised objects and which may not; and (ii) a type checker, which can be integrated into the BCV, to statically check the program at load time. To validate our approach, we have followed the methodology that we announce in the introduction of this chapter. First we formalised our type system and machine checked its soundness proof using the Coq proof assistant for an idealised object-oriented language. Then we implemented a prototype for the full Java bytecode language and experimentally validated our solution on a large number of classes from Oracle's Java Runtime Environment (JRE).

This work has been published in [HJMP10]. In this document, we restrict our presentation to the overall motivations of this work. The type system we provide is inspired by previous works on null pointer analyses by Fähndrich and Leino [FL03], and by a previous formalisation of their type system by Hubert, Jensen and Pichardie [HJP08b].

4.2.1 Current Limits of the BCV and Current Oracle's Recommendations

Fig. 4.2 is an extract of class `ClassLoader` of SUN's JRE as it was before 1997. The security policy which needs to be ensured is that `resolveClass`, a security sensitive method, may be

```
1 public abstract class ClassLoader {
2     private ClassLoader parent;
3     protected ClassLoader() {
4         SecurityManager sm = System.getSecurityManager();
5         if (sm != null) {sm.checkCreateClassLoader();}
6         this.parent = ClassLoader.getSystemClassLoader();
7     }
8     protected final native void resolveClass(Class c);
9 }
```

Figure 4.2: Extract of the ClassLoader of Oracle's JRE

called only if the security check at line 5 has succeeded. To ensure this security property, this code relies on the properties enforced on object initialization by the BCV.

In Java, objects are initialised by calling a class-specific constructor which is supposed to establish an invariant on the newly created object. The BCV enforces two properties related to these constructors: Before accessing an object, (i) a constructor of its dynamic type must have been called and (ii) each constructor either calls another constructor of the same class or a constructor of the super-class on the object under construction, except for `java.lang.Object` which has no super-class. This implies that it is not possible to bypass a level of constructor. To deal with exceptional behaviour during object construction, the BCV enforces another property — concisely described in *The Java Language Specification* [GJSB05], Section 12.5, or implied by the type system described in the JSR202 [Buc06]): If one constructor finishes abruptly, then the whole construction of the object finishes abruptly. Thus, if the construction of an object finishes normally, then all constructors called on this object have finished normally. Failure to implement this verification properly led to a famous attack [DFW96] in which it was exploited that if code such as `try {super();} catch(Throwable e){}` in a constructor is not rejected by the BCV, then malicious classes can create security-critical classes such as class loaders.

However, even with these two properties enforced, it is not guaranteed that uninitialised objects cannot be used. In Fig. 4.2, if the check fails, the method `checkCreateClassLoader` throws an exception and therefore terminates the construction of the object, but the garbage collector then call a `finalize()` method, which is an instance method and has the object to be collected as receiver (cf. Section 12.6 of [GJSB05]). An attacker could code another class that extends `ClassLoader` and has a `finalize()` method. If run in a right-restricted context, *e.g.* an applet, the constructor of `ClassLoader` fails (at line 5) but later the garbage collector calls the attacker's `finalize` method. The attacker can therefore call the `resolveClass` method on it, bypassing the security check in the constructor and breaking the security of Java.

The initialization policy enforced by the BCV is in fact too weak: when a method is called on an object, there is no guarantee that the construction of an object has been successfully run. An ad-hoc solution to this problem is proposed by SUN [Sun10] in its Guideline 4-3 *Defend against partially initialised instances of non-final classes*: adding a special boolean field to each class for which the developer wants to ensure it has been sufficiently initialised. This field, set to **`false`** by default, should be private and should be set to **`true`** at the end of the constructor. Then, every method that relies on the invariant established by the constructor must test whether this field is set to **`true`** and fail otherwise. If `initialised` is **`true`**, the construction of the object up to the initialization of `initialised` has succeeded. Checking if `initialised` is **`true`** allows to ensure that sensitive code is only executed on classes that have

```
public abstract class ClassLoader {
    private volatile boolean initialised;
    private ClassLoader parent;
    protected ClassLoader() {
        SecurityManager sm = System.getSecurityManager();
        if (sm != null) {sm.checkCreateClassLoader();}
        this.parent = ClassLoader.getSystemClassLoader();
        this.initialised = true;
    }
    private void check() {
        if (!initialised)
            throw new SecurityException("ClassLoader object not initialised");
    }
    protected final void resolveClass(Class c){
      this.check();
      this.resolveClass0(c);
    }
    private native void resolveClass0(Class c);
}
```

Figure 4.3: Extract of the ClassLoader of Oracle's JRE

been initialised up to the constructor of the current class. Fig. 4.3 shows the same extract as in Fig. 4.2 but with the needed instrumentation (this is the current implementation as of JRE 1.6.0_16).

Although there are some exceptions and some methods are designed to access partially initialised objects (for example to initialize the object), most methods should not access partially initialised objects. Following the remediation solution proposed in the CERT's recommendation or Oracle's guideline 4-3, a field should be added to almost every class and most methods should start by checking this field. This is resource consuming and error prone because it relies on the programmer to keep track of what is the semantic invariant, without providing the adequate automated software development tools. It may therefore lead not to functional bugs but to security breaches, which are harder to detect. In spite of being known since 1997, this pattern is not always correctly applied to all places where it should be. This has lead to security breaches, see *e.g.*, the Secunia Advisory SA10056 [Sec03].

4.2.2 The Right Way: A Type System

We propose a twofold solution: first, a way to specify the security policy which is simple and modular, yet more expressive than a single boolean field; second, a modular type checker, which could be integrated into the BCV, to check that the whole program respects the policy.

We require every Java program to add type annotation to existing type declarations (fields and method signatures). These type annotation follow the grammar[1]:

$$\mathit{Type} ::= \mathit{Init} \mid \mathit{Raw} \mid \mathit{Raw}(C)$$

We introduce two main types: *Init*, which specifies that a reference can only point to a fully initialised object or the `null` constant, and *Raw*, which specifies that a reference may point to a partially initialised object. A third annotation, $\mathit{Raw}(C)$ (with C a class name) allows to precise that the object may be partially initialised but that all constructors up to and including the constructor of C must have been fully executed. *E.g.*, when one checks that

[1]In our implementation, we rely on Java annotations in order to easily extend the grammar of Java types.

`initialised` contains **`true`** in `ClassLoader.resolveClass`, one checks that the receiver has the type *Raw*(`ClassLoader`). Those type annotation must also be given for the implicit argument `this` of every instance method.

We introduce a `@Pre` Java annotation to specify the type of the receiver at the beginning of the method. Some methods, usually called from constructors, are meant to initialize their receiver. We have therefore added the possibility to express this by adding a `@Post` annotation for the type of the receiver at the end of the method. These annotations take as argument an initialization level produced by the rule `V_ANNOT`.

Another part of the security policy is the `SetInit` instruction, which mimics the instruction `this.initialised = true` in Oracle's guideline. It is implicitly put at the end of every constructor but it can be explicitly placed before. It declares that the current object has completed its initialization up to the current class. Note that the object is not yet considered fully initialised as it might be called as a parent constructor in a subclass.

Fig. 4.4 shows class `ClassLoader` with its policy specification. The policy ensured by the current implementation of Oracle is slightly weaker: it does not ensure that the receiver is fully initialised when invoking `resolveClass` but simply checks that the constructor of `ClassLoader` has been fully run. On this example, we can see that the constructor has the annotations `@Pre(`*Raw*`)`, meaning that the receiver may be completely uninitialised at the beginning, and `@Post(`*Raw*`(ClassLoader))`, meaning that, on normal return of the method, at least one constructor for each parent class of `ClassLoader` and a constructor of `ClassLoader` have been fully executed.

```
public abstract class ClassLoader {
    Init private ClassLoader parent;

    @Pre(Raw) @Post(Raw(ClassLoader)) protected ClassLoader() {
        SecurityManager sm = System.getSecurityManager();
        if (sm != null) {sm.checkCreateClassLoader();}
        this.parent = ClassLoader.getSystemClassLoader();
    }
    @Pre(Init) @Post(Init) protected final native void resolveClass(Init Class c);
}
```

Figure 4.4: Extract of the ClassLoader of Oracle's JRE

We formalise this type system on a simple language in-between Java source and Java bytecode. Our goal was to have a language close enough to the bytecode in order to easily obtain, from the specification, a naive implementation at the bytecode level while keeping a language easy to reason with. It is based on the IR language that we provided in Section 4.1. In our formalisation, we specially benefit from the lack of operand stack and the fact that, contrary to Java bytecode, it is the same instruction that allocates the object and calls the constructor.

The semantic of the language is standard except we track in each object, its initialization status. On top of this semantics, we formalised an proved correct a type soundness theorem. It states that every time a method call is performed, its actual argument (including the receiver) have an initialization status compatible with the type annotations of the called method.

4.2.3 Experimental Validation on Oracle's JRE

In order to show that our type system allows to verify legacy code with only a few annotations, we implemented a standalone prototype, handling the full Java bytecode, and we tested all classes of packages `java.lang`, `java.security` and `javax.security` of the JRE1.6.0_20.

To keep the burden of type annotation low, we defined as default values the most precise type that may be use in each context. This gives a *safe by default* policy. By default,

- Fields, method parameters and return values are fully initialised objects (written *Init*).
- Constructors take a receivers uninitialised at the beginning (`@Pre(`*Raw*`)`) and initialised up-to the current class at the end (written `@Post(`*Raw*`(C))` if in the class `C`).
- Other methods take a receiver fully initialised (`@Pre(`*Init*`)`).
- Except for constructors, method receivers have the same type at the end as at beginning of the method (written `@Post(A)` if the method has the annotation `@Pre(A)`).

If we remove from Fig. 4.4 the default annotations, we obtain the original code in Fig. 4.2. It shows that despite choosing the strictest (and safest) initialization policy as default, the annotation burden can be kept low.

In our experimentation, 348 classes out of 381 were proven safe *w.r.t.* the default policy without any modification. By either specifying the actual policy when the default policy was too strict, or by adding cast instructions (see [HJMP10]) when the type system was not precise enough, we were able to verify 377 classes, that is to say 99% of classes. We discuss in [HJMP10] the 4 remaining classes that are not yet proven correct by our analysis. The modifications represent only 55 source lines of code out of 131,486 for the three packages studied. Moreover most code modifications are to express the actual initialization policy, which means existing code can be proven safe. Only 45 methods out of 3,859 (1.1%) and 2 fields out of 1,524 were annotated. Last but not least, the execution of the type checker takes less than 20 seconds for the packages studied.

Our prototype, the Coq formalization and proofs, as well as examples of annotated classes can be found at

`http://www.irisa.fr/celtique/ext/rawtypes`

4.3 Static Enforcement of Policies for Secure Object Copying

Exchanging data objects with untrusted code is a delicate matter because of the risk of creating a data space that is accessible by an attacker. Consequently, secure programming guidelines for Java such as those proposed by Oracle [Ora10] and CERT [CER10b] stress the importance of using defensive *copying* or *cloning* before accepting or handing out references to an internal mutable object. There are two aspects of the problem:

1. If the result of a method is a reference to an internal mutable object, then the receiving code may modify the internal state. Therefore, it is recommended to make copies of mutable objects that are returned as results, unless the intention is to share state.
2. If an argument to a method is a reference to an object coming from hostile code, a local copy of the object should be created. Otherwise, the hostile code may be able to modify the internal state of the object.

A common way for a class to provide facilities for copying objects is to implement a `clone`() method that overrides the cloning method provided by `java.lang.Object`. The following code snippet, taken from Oracle's Secure Coding Guidelines for Java, demonstrates how a `date` object is cloned before being returned to a caller:

```
public class CopyOutput {
    private final java.util.Date date;
    ...
    public java.util.Date getDate() {
        return (java.util.Date)date.clone();
    }
}
```

However, relying on calling a polymorphic `clone` method to ensure secure copying of objects may prove insufficient, for two reasons. First, the implementation of the `clone`() method is entirely left to the programmer and there is no way to enforce that an untrusted implementation provides a sufficiently *deep* copy of the object. It is free to leave references to parts of the original object being copied in the new object. Second, even if the current `clone`() method works properly, sub-classes may override the `clone`() method and replace it with a method that does not create a sufficiently deep clone. For the above example to behave correctly, an additional class invariant is required, ensuring that the `date` field always contains an object that is of class `Date` and not one of its sub-classes. To quote from the CERT guidelines for secure Java programming: *"Do not carry out defensive copying using the clone() method in constructors, when the (non-system) class can be subclassed by untrusted code. This will limit the malicious code from returning a crafted object when the object's clone() method is invoked."* Clearly, we are faced with a situation where basic object-oriented software engineering principles (sub-classing and overriding) are at odds with security concerns. To reconcile these two aspects in a manner that provides semantically well-founded guarantees of the resulting code, this work proposes a formalism for defining *cloning policies* by annotating classes and specific copy methods, and a static enforcement mechanism that will guarantee that all classes of an application adhere to the copy policy. We do not enforce that a copy method will always return a target object that is functionally equivalent to its source. Rather, we ensure non-sharing constraints between source and targets, expressed through a copy policy, as this is the security-critical part of a copy method in a defensive copying scenario.

4.3.1 Background: Cloning of Objects

For objects in Java to be cloneable, their class must implement the empty interface `Cloneable`. A default `clone` method is provided by the class `Object`: when invoked on an object of a class, `Object.clone` will create a new object of that class and copy the content of each field of the original object into the new object. The object and its clone share all sub-structures of the object; such a copy is called *shallow*.

It is common for cloneable classes to override the default clone method and provide their own implementation. For a generic `List` class, this could be done as follows:

```
public class List<V> implements Cloneable
{
    public V value;
    public List<V> next;

    public List(V val, List<V> next) {
        this.value = val;
        this.next = next; }
    public List<V> clone() {
        return new List(value,(next==null) ? null : next.clone()); }
}
```

Notice that this cloning method performs a shallow copy of the list, duplicating the spine but sharing all the elements between the list and its clone. Because this amount of sharing may not be desirable (for the reasons mentioned above), the programmer is free to implement other versions of `clone()`. For example, another way of cloning a list is by copying both the list spine and its elements[2], creating what is known as a *deep* copy.

```
public List<V> deepClone() {
  return new List((V) value.clone(), (next==null) ? null : next.deepClone()); }
```

A general programming pattern for methods that clone objects works by first creating a shallow copy of the object by calling the `super.clone()` method, and then modifying certain fields to reference new copies of the original content. This is illustrated in the following snippet, taken from the class `LinkedList` in Fig. 4.6:

```
public Object clone() {  ...
  clone = super.clone();  ...
  clone.header = new Entry<E>(null, null, null); ...
  return clone;}
```

There are two observations to be made about the analysis of such methods. First, an analysis that tracks the depth of the clone being returned will have to be flow-sensitive, as the method starts out with a shallow copy that is gradually being made deeper. This makes the analysis more costly. Second, there is no need to track precisely modifications made to parts of the memory that are not local to the clone method, as clone methods are primarily concerned with manipulating memory that they allocate themselves. This will have a strong impact on the design choices of our analysis.

4.3.2 Copy Policies

The first contribution of this work is a proposal for a set of semantically well-defined program annotations, whose purpose is to enable the expression of policies for secure copying of objects. Introducing a copy policy language enables class developers to state explicitly the intended behavior of copy methods. In the basic form of the copy policy formalism, fields of classes are annotated with `@Shallow` and `@Deep`. Intuitively, the annotation `@Shallow` indicates that the field is referencing an object, parts of which may be referenced from elsewhere. The annotation `@Deep(X)` on a field `f` means that *a*) the object referenced by this field `f` cannot itself be referenced from elsewhere, and *b*) the field `f` is copied according to the copy policy identified by `X`. Here, `X` is either the name of a specific policy or if omitted, it designates the default policy of the class of the field. For example, the following annotations:

```
  class List  { @Shallow V value;  @Deep List next;  ...}
```

specifies a default policy for the class `List` where the `next` field points to a list object that also respects the default copy policy for lists. Any method in the `List` class, labelled with the `@Copy` annotation, is meant to respect this default policy.

In addition it is possible to define other copy policies and annotate specific *copy methods* (identified by the annotation `@Copy(...)`) with the name of these policies. For example, the

[2]To be type-checked by the Java compiler it is necessary to add a cast before calling `clone()` on `value`. A cast to a sub interface of `Cloneable` that declares a `clone()` method is necessary.

annotation[3]

```
DL: { @Deep V value; @Deep(DL) List next;};
@Copy(DL) List<V> deepClone() {
  return new List((V) value.clone(),
                  (next==null ? null : next.deepClone())); }
```

can be used to specify a list-copying method that also ensures that the `value` fields of a list of objects are copied according to the copy policy of their class (which is a stronger policy than that imposed by the annotations of the class `List`).

The annotations are meant to ensure a certain degree of non-sharing between the original object being copied and its clone. We want to state explicitly that the parts of the clone that can be accessed via fields marked @`Deep` are unaccessible from any part of the heap that was accessible before the call to `clone`(). The presentation is kept informal here but to make this intention precise, we provide in [JKP11, JKP12] a formal semantics of a simple programming language extended with policy annotations and define what it means for a program to respect a policy.

4.3.3 Enforcement

The second contribution of this work is to make the developer's intent, expressed by copy policies, statically enforceable using a type system. We formalize this enforcement mechanism by giving an interpretation of the policy language in which annotations are translated into graph-shaped type structures. For example, the annotations of the `List` class defined above will be translated into the graph that is depicted to the right in Fig. 4.5 (`res` is the name given to the result of the copy method). The left part shows the concrete heap structure.

Unlike general purpose shape analysis, we take into account the programming methodologies and practice for copy methods, and design a type system specifically tailored to the enforcement of copy policies. This means that the underlying analysis must be able to track precisely all modifications to objects that the copy method allocates itself (directly or indirectly) in a flow-sensitive manner. Conversely, as copy methods should not modify non-local objects, the analysis will be designed to be more approximate when tracking objects external to the method under analysis, and the type system will accordingly refuse methods that attempt such non-local modifications. As a further design choice, the annotations are required to be verifiable modularly on a class-by-class basis without having to perform an analysis of the entire code base, and at a reasonable cost.

As depicted in Fig. 4.5, concrete memory cells are either abstracted as *a*) $\top_{out}$ when they are not allocated in the copy method itself (or its callee); *b*) $\top$ when they are just marked as *maybe-shared*; and *c*) circle nodes of a deterministic graph when they are locally allocated. A single circle furthermore expresses a singleton concretization. We use otherwise a double circle. In the first case, the type system will be able to perform destructive updates of the graph structure (strong update). In the latter, only a conservative update is sound (weak update). In this example, the abstract heap representation matches the graph interpretation of annotations, which means that the instruction set that produced this heap state satisfies the specified copy policy.

Technically, the intra-procedural component of our analysis corresponds to heap shape analysis with the particular type of graphs that we have defined. Operations involving non-

[3]Our implementation uses a sightly different policy declaration syntax because of the limitations imposed by the Java annotation language.

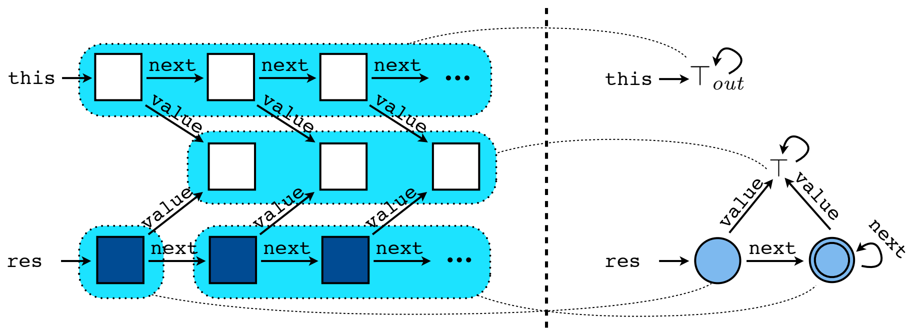


Figure 4.5: A linked structure (left part) and its abstraction (right part).

local parts of the heap are rapidly discarded. Inter-procedural analysis uses the signatures of copy methods provided by the programmer. Inheritance is dealt with by stipulating that inherited fields retain their "shallow/deep" annotations. Redefinition of a method must respect the same copy policy and other copy methods can be added to a sub-class. The detailed definition of the analysis is presented in in [JKP11, JKP12] as a set of type inference rules for simple core Java langage (in the same spirit than in Section 4.2). We prove in Coq a soundness theorem which states that if all copy methods of a program are typable then they all respects there copy specification.

Note that the type system if flow-sensitive and in order to type-check a method, it is necessary to infer intermediate types at each loop header and conditional junction points. By turning the typing problem into a fixpoint problem in a suitable sup-semi-lattice structure, we automatically infer those types. A widening operator is also introduced to prevent infinite ascending iterations.

Figure 4.6 demonstrates the use of the type system on a challenging example taken from the standard Java library. The class `java.util.LinkedList` provides an implementation of doubly-linked lists. A list is composed of a first cell that points through a field `header` to a collection of doubly-linked cells. Each cell has a link to the previous and the next cell and also to an element of (parameterized) type `E`. The clone method provided in `java.lang` library implements a shallow copy where only cells of type `E` may be shared between the source and the result of the copy. In Fig. 4.6 we present a modified version of the original source code: we have inlined all method calls, except those to copy methods and removed exception handling that leads to an abnormal return from methods[4]. Note that one method call in the original code was virtual and hence prevented inlining. Is has been necessary to make a private version of this method. This makes sense because such a virtual call actually constitutes a potentially dangerous hook in a cloning method, as a re-defined implementation could be called when cloning a subclass of `Linkedlist`.

In Fig. 4.6 we provide several intermediate types that are necessary for typing this method (T_i is the type before executing the instruction at line i). The call to `super.clone` at line 12 creates a shallow copy of the header cell of the list, which contains a reference to the original list. The original list is thus shared, a fact which is represented by an edge to $\top_{out}$ in type T_{13}.

The copy method then progressively constructs a deep copy of the list, by allocating

[4]Inlining is automatically performed by our tool and exception control flow graph is managed as standard control flow but omitted here for simplicity.

```
 1 class LinkedList<E> implements Cloneable {
 2   private @Deep Entry<E> header;
 3
 4   private static class Entry<E> {
 5     @Shallow E element;
 6     @Deep Entry<E> next;
 7     @Deep Entry<E> previous;
 8   }
 9
10   @Copy public Object clone() {
11     LinkedList<E> clone = null;
12     clone = (LinkedList<E>) super.clone();
13     clone.header = new Entry<E>;
14     clone.header.next = clone.header;
15     clone.header.previous = clone.header;
16     Entry<E> e = this.header.next;
17     while (e != this.header) {
18       Entry<E> n = new Entry<E>;
19       n.element = e.element;
20       n.next = clone.header;
21       n.previous = clone.header.previous;
22       n.previous.next = n;
23       n.next.previous = n;
24       e = e.next;
25     }
26     return clone;
27   }
28 }
```

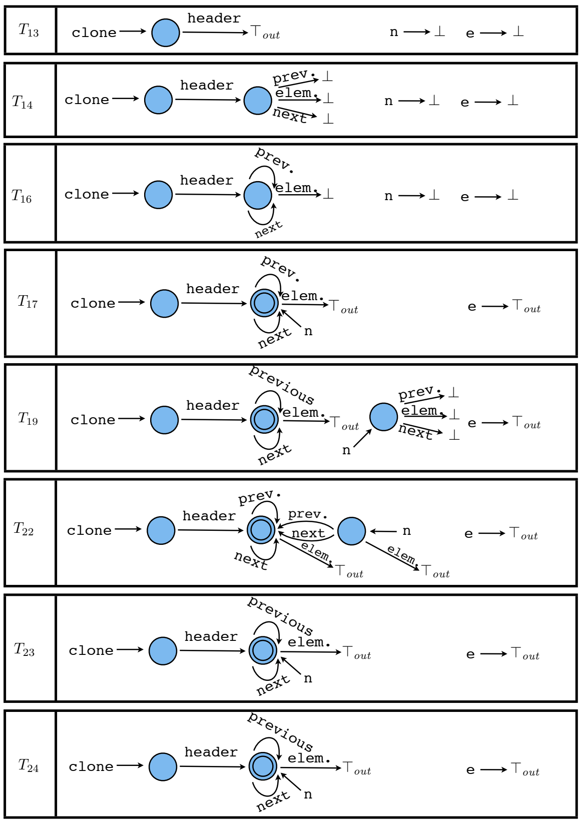


Figure 4.6: Intermediate Types for `java.util.LinkedList.clone()`

a new node (see type T_{14}) and setting all paths `clone.header`, `clone.header.next` and `clone.header.previous` to point to this node. This is reflected in the analysis by a *strong update* to the node representing path `clone.header` to obtain the type T_{16} that precisely models the alias between paths `clone.header`, `clone.header.next` and `clone.header.previous` (the Java syntax used here hides the temporary variable that is introduced to be assigned the value of `clone.header` and then be updated).

This type T_{17} is the loop invariant necessary for type checking the whole loop. It is a super-type of T_{16} (updated with $e \mapsto \top_{out}$) and of T_{24} which represents the memory at the end of the loop body. The body of the loop allocates a new list cell (pointed to by variable `n`) (see type T_{19}) and inserts it into the doubly-linked list. The assignment in line 22 updates the weak node pointed to by path `n.previous` and hence merges the strong node pointed to by `n` with the weak node pointed to by `clone.header`, representing the spine of the list. The assignment at line 23 does not modify the type T_{23}.

Notice that the types used in this example show that a flow-insensitive version of the analysis could not have found this information. A flow-insensitive analysis would force the merge of the types at all program points, and the call to `super.clone` return a type that is less precise than the types needed for the analysis of the rest of the method.

4.3.4 Experiments

The policy language and its enforcement mechanism has been implemented in the form of a security tool for Java byte code. Copying policies are expressed using Java annotations. Both the policy extraction and enforcement components are implemented using the Sawja platform.

In its standard mode, the tool performs a modular verification of annotated classes. We have run experiments on some classes in the package `java.util` of the standard library. We have successfully checked some realistic copy signatures for them (doubly-linked lists, hash tables). However, some cloning methods will necessarily be beyond the reach of the analysis. We have identified one such method in GNU Classpath's `TreeMap` class, where the merging of information at control flow merge points destroys too much of the inferred type graph. A disjunctive form of abstraction seems necessary to verify a deep copy annotation on such programs and we leave this as a challenging extension.

The analysis is also capable of processing un-annotated methods, albeit with less precision than when copy policies are available—this is because it cannot rely on annotations to infer external copy method types. Nevertheless, this capability allows us to test our tool on two large code bases. The 17000 classes in Oracle's `rt.jar` and the 7000 in the GNU Classpath have passed our scanner un-annotated. Among the 459 `clone()` methods we found in these classes, we were only unable to infer the minimal signatures $\{\}$ (the same signature as `java.lang.Object.clone()`) in 93 methods. The other methods all provide a signature with some deep fields. Our prototype confirms the efficiency of the enforcement technique because all these verifications took only 25s on a laptop computer.

Our prototype, the Coq formalization and proofs, as well as examples of annotated classes can be found at

http://www.irisa.fr/celtique/ext/clones

4.4 Related Work

Object Initialization in Java Object initialization has been studied from different points of view. Freund and Mitchell [FM03] have proposed a type system that formalizes and enforces the initialization properties ensured by the BCV, which are not sufficient to ensure that no partially initialized object is accessed. Unlike local variables, instance fields have a default value (`null`, `false` or `0`) which may be then replaced by the program. The challenge is then to check that the default value has been replaced before the first access to the field (*e.g.* to ensure that all field reads return a non-null value). This is has been studied in its general form by Fähndrich and Xia [FX07], and Qi and Myers [QM09]. Those works are focused on enforcing invariants on fields and finely tracks the different fields of an object. They also try to follow the objects after their construction to have more information on initialized fields. This is an overkill in our context. Unkel and Lam studied another property of object initialization: stationary fields [UL08]. A field may be stationary if all its reads return the same value. There analysis also track fields of objects and not the different initialization of an object. In contrast to our analysis, they stop to track any object stored into the heap.

Enforcement of Copying Policy in Java To the best of our knowledge, the current work is the first to propose a formal, semantically founded framework for secure cloning through program annotation and static enforcement. The closest work in this area is that of Anderson *et al.* [AGN09] who have designed an annotation system for C data structures in order to control sharing between threads. Annotation policies are enforced by a mix of static and run-time verification. On the run-time verification side, their approach requires an operator that can dynamically “cast” a cell to an unshared structure. In contrast, our approach offers a completely static mechanism with statically guaranteed alias properties.

Aiken *et al.* proposes an analysis for checking and inferring local non-aliasing of data [AFKT03]. They propose to annotate C function parameters with the keyword `restrict` to ensure that no other aliases to the data referenced by the parameter are used during the execution of the method. A type and effect system is defined for enforcing this discipline statically. This analysis differs from ours in that it allows aliases to exist as long as they are not used whereas we aim at providing guarantees that certain parts of memory are without aliases. The properties tracked by our type system are close to escape analysis [Bla99, CGS$^+$99] but the analyses differ in their purpose. While escape analysis tracks locally allocated objects and tries to detect those that do not escape after the end of a method execution, we are specifically interested in tracking locally allocated objects that escape from the result of a method, as well as analyse their dependencies with respect to parameters.

Our static enforcement technique falls within the large area of static verification of heap properties. A substantial amount of research has been conducted here, the most prominent being region calculus [TT97], separation logic [OYR04] and shape analysis [SRW02]. Of these three approaches, shape analysis comes closest in its use of shape graphs. Shape analysis is a large framework that allows to infer complex properties on heap allocated data-structures like absence of dangling pointers in C or non-cyclicity invariants. In this approach, heap cells are abstracted by shape graphs with flexible object abstractions. Graph nodes can either represent a single cell, hence allowing strong updates, or several cells (summary nodes). *Materialization* allows to split a summary node during cell access in order to obtain a node pointing to a single cell. The shape graphs that we use are not intended to do full shape analysis but are rather specialized for tracking sharing in locally allocated objects. We use a different naming strategy for graph nodes and discard all information concerning non-locally allocated references. This leads to an analysis which is more scalable than full shape analysis, yet still powerful enough for verifying complex copy policies as demonstrated in the concrete case study `java.util.LinkedList`.

4.5 Conclusions and Perspectives

The Sawja framework we have presented in Section 4.1 has been a powerful tool when working on the static analyses presented in Section 4.2 and 4.3. It provides features that accelerate the rapid prototyping of these analyses. The IR that are provided by the framework are of great help. Our formalisation effort has been performed with the Coq proof assistant on a small object-oriented language. But as we understand better the semantic foundations of this IR we also better understand the gap between what is formalised in Coq and what is implemented in OCaml.

In term of proof effort, the mechanized proofs that have been performed here are far simpler than in the previous chapter. Each of them took less then 2 man-months. It means we have more time to really think about the design of the analysis itself. Of course the analyses we run at the end are not fully verified in the sens of Chapter 2 and 3 but they are nevertheless validated in a very complementary and original way.

None of the components of the Sawja library are currently verified but several of them have been designed with this long-term goral in mind.

- Verified parsing is not our primarily goal. We believe this component have to be trusted and carefully tested. Still the recent work of Jourdan et al. [JPL12] may be considered to verify this part.

- Decompilation from bytecode to a higher level intermediate representation is the part that has attracted most of our formal effort. Mechanising this proof should be routine work as our paper-proof is rather detailed.

  We believe intermediate representation (IR) are the missing formal link between a tool that run on real programs and an idealized core language on which we would like to perform meta-reasoning. Delphine Demange's PhD work has been dedicated to this topic.

  Still our implementation relies on sub-components that are out of scope of the current formalisation. First, we perform an inlining of subroutines (bytecodes `jsr`/`ret`). Subroutines have been pointed out by the research community as raising major static analysis difficulties [SA98]. Such a transformation is non-trivial to be proved correct but this language feature is progressively disappearing in the Java bytecode format anyway. Second, our intermediate representation rebuilds types using a static analysis similar to the bytecode verifier. This component has been deeply formalised by the research community but integrating such a work in our verified platform would certainly require some times.

- Sharing of objects is done at parsing time when building the first internal representation of programs. The current implementation relies on imperative hash tables to associate an unique number to every syntactic term. We should be able to verify a similar mechanism using the various map structures that are available in the Coq library.

Chapter 5

Verified Static Single Assignment Intermediate Representation

The works we have reported in the last chapter show the importance of intermediate representation in a verified software toolchains. Among the existing toolchains, the Static single assignment (SSA) is a leading form, specially in modern compilers, including GCC and LLVMC [LLV], but also in static analysis platforms [Cho, IBM]. SSA form [CFR$^+$91] is an intermediate representation where variables are statically assigned exactly once. Thanks to the considerable strength of this property, the SSA form simplifies the definition of many optimizations, and improves their efficiency, as well as the quality of their results. It is therefore not surprising that there is a vast body of work on SSA. However, the simplicity of SSA form is deceptive, and designing a correct SSA-based middle-end compiler has been fraught with difficulties [BCHS98].

The content of this chapter is more specifically oriented toward compilers than the previous chapters. Indeed our formalisation has been conducted in the context of the CompCert verified compiler. Leroy's CompCert has been rightfully acclaimed as a *tour de force*, but it foregoes relying on an SSA-based middle end. In [Ler09b], Leroy reports:

> Since the beginning of CompCert we have been considering using SSA-based intermediate languages, but were held off by two difficulties. First, the dynamic semantics for SSA is not obvious to formalize. Second, the SSA property is global to the code of a whole function and not straightforward to exploit locally within proofs.

The thesis of our work is that a compiler or a static analysis can be realistic, verified and still rely on a SSA form. To support our thesis, we provide the first verified SSA-based middle-end. Rather than programming and proving a verified compiler from scratch, we have programmed and verified a SSA-based middle-end compiler that can be plugged into CompCert at the level of RTL. Fig. 5.1 describes the overall architecture. Our middle-end performs four phases: (i) normalization of RTL program; (ii) transformation from RTL form into SSA form; (iii) optimization of programs in SSA form, including Global Value Numbering (GVN) [AWZ88]; (iv) transformation of programs from SSA form to RTL form; and relies on CompCert for the transformation from C to RTL programs prior to SSA conversion, and from RTL programs to assembly code after conversion out of SSA—our point is to program a realistic and verified SSA-based middle-end, rather than to demonstrate that SSA-based optimizations dramatically improve the efficiency of generated code.

We validate our compiler middle-end with a mix of techniques directly inherited from CompCert. We resort to translation validation [PSS98, Nec00]—increasingly favored by CompCert [TL09, TL10]—for converting programs into SSA form and for GVN. Specifically, we

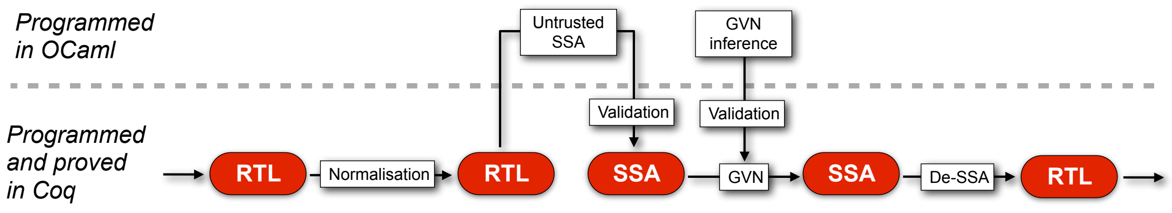


Figure 5.1: The SSA Middle-end

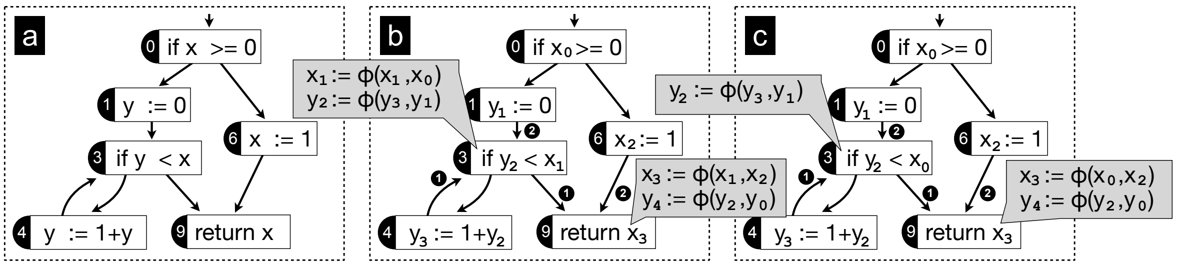


Figure 5.2: Example programs. Programs b), c) are SSA forms of a). Programs b) is in naive form and c) in minimal form.

program in Coq verified checkers that validate *a posteriori* results of untrusted computations, and we implement in OCaml efficient algorithms for these computations; we rely on Cytron *et al* algorithm [CFR+91] for computing minimal SSA form, and on Alpern *et al* iteration strategy [AWZ88] for computing a numbering in GVN. In contrast, the normalization of the RTL program, and the conversion out of SSA are directly programmed and proved in Coq. In addition, our work addresses the two issues raised by Leroy [Ler09b]. First, we give a simple and intuitive operational semantics for SSA; the semantics follows the informal description given in [CFR+91], and does not require any artificial state instrumentation. Second, we define on SSA programs two global properties, called strictness and equational form, allowing to conclude reasonably directly that the substitutions performed by GVN and other optimizations are sound.

Summarizing, our work provides the first verified SSA-based middle-end, the first formal proof of an SSA-based optimization, as well as an intuitive semantics for SSA. It thus serves as a good starting point for further studies of verified and realistic SSA-based compilers. This chapter is heavily based on [BDP12] but omits some technical details about the type system we use to validate the SSA transformation.

5.1 Background

Static Single Assignment form is an intermediate representation in which variables are statically assigned exactly once, thus making explicit in the program syntax the link between the program point where a variable is defined and read.

Converting into SSA form is easy for straighline code: one simply tags each variable definition with an index, and each variable use with the index corresponding to the last definition of this variable. For example, $[x := 1; y := x + 1; x := y - 1; y := x]$ is transformed into $[x_0 := 1; y_0 := x_0 + 1; x_1 := y_0 - 1; y_1 := x_1]$. The transformation is semantics-preserving, in the sense that the final values of x and y in the first snippet coincide with the final values of x_1 and y_1 in the second snippet. On the other hand, one cannot transform arbitrary programs into semantically equivalent programs in SSA form solely by tagging variables: one must insert *ϕ-functions* to handle branching statements. Fig. 5.2 shows a program a), and a program b) that corresponds to a SSA form of a). In program a), the value of variable x read at node 9 either comes from the definition of x at entry or at node 6. In program b), these two definitions of x are renamed into the unique definition of x_0 and x_2 and merged together by the ϕ-function of x_3 at entry of node 9. The precise meaning of a ϕ-block depends on the numbering convention of the predecessor nodes of each junction point. In Fig. 5.2 b) we make explicit this numbering by labelling the CFG edges. For example, node 3 is the first predecessor of point 9 and node 6 is the second one. The semantics of ϕ-functions is given in the seminal paper by Cytron *et al* [CFR+91]:

"If control reaches node j from its kth predecessor, then the run-time support remembers k while executing the ϕ-functions in j. The value of $\phi(x_1, x_2, \ldots)$ is just the value of the kth operand. Each execution of a ϕ-function uses only one of the operands, but which one depends on the flow of control just before entering j. "

There may be several SSA forms for a single control-flow graph program; Fig. 5.2 b) and c) gives alternative SSA forms for program a). As the number of ϕ-functions directly impacts the quality of the subsequent optimizations—as well as the size of the SSA form—it is important that SSA generators for real compilers produce a SSA form with a minimal number of ϕ-functions. Implementations of minimal SSA generally rely on the notion of *dominance frontier* to choose where to insert ϕ-functions. A node i in a CFG dominates another node j if every path from then entry of the CFG to j contains i. The dominance is said to be strict if additionally $i \neq j$. A tree can encode the dominance relation between the nodes of the CFG. For a node i of a CFG, the *dominance frontier* $DF(i)$ of i is defined as the set of nodes j such that i dominates at least one predecessor of j in the CFG but does not strictly dominates j itself. The notion is extended to a set of nodes S with $DF(S) = \bigcup_{i \in S} DF(i)$. The *iterated dominance frontier* $DF^+(S)$ of a set of nodes S is $\lim_{i \to \infty} DF^i(S)$, where $DF^1(S) = DF(S)$ and $DF^{i+1}(S) = DF(S \cup DF^i(S))$. Formally, a program is in *minimal-SSA* form when a ϕ-function of an instance x_i of an original variable x appears in a junction point j iff j belongs to the iterated dominance frontier of the set of definition nodes of x in the original program. For instance, program c) in Fig. 5.2 is in minimal-SSA form. However, one can achieve more compact SSA forms by observing that, at any junction point, dead variables need not be defined by a ϕ-function. The intuition is captured by the notion of *pruned-SSA* form: a program is in *pruned-SSA* form if it is in minimal-SSA form and for each ϕ-function of an instance x_i of an original variable x at a junction point j, x is live at j in the original program (there is a path from j to a use of x that does not redefine x).

SSA-based optimizations The SSA form simplifies the definition of many common optimizations; for instance, copy propagation algorithms can just walk through a SSA program, identify statements of the form $x := y$, and replace every use of x by y. Furthermore, several optimizations are naturally formulated on SSA. One typical SSA-based optimization is *Global Value Numbering* (GVN) [AWZ88], which assigns to variables an identifying number such that variables with the same number will hold equal values at execution time. The effectiveness of GVN lies in its ability to compute efficiently numberings that identify as many variables as possible. Advanced algorithms [AWZ88, BCS97] allow to compute efficiently such numberings. We briefly explain one such numbering in Section 5.5.

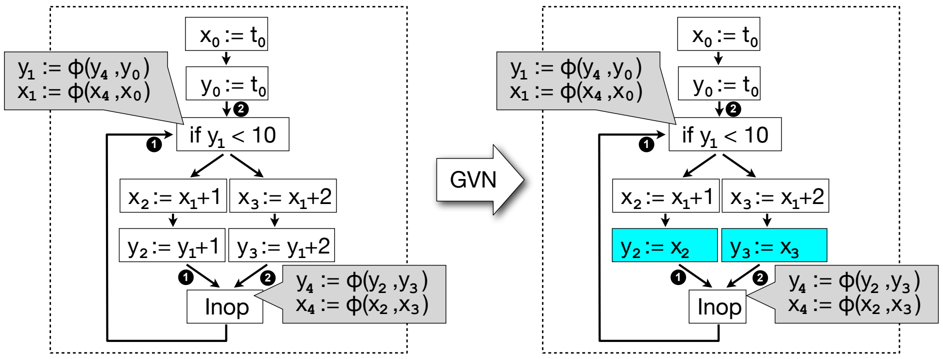


Figure 5.3: Common sub-expression elimination (CSE) using GVN

Fig. 5.3 illustrates how GVN can be used to eliminate redundant computation. The left program is the original code; in this program, for each i, x_i and y_i are assigned the same value number. Hence, the evaluation of $y_1 + 1$ (resp. $y_1 + 2$) is a redundant computation when assigning y_2 (resp. y_3), and one can transform the program into the semantically equivalent one shown on the right of the figure. The strength of the analysis lies in its ability to reason about ϕ-functions, which allows it to infer the equality $x_2 = y_2$. This is only possible because numbering is global to the whole program; in fact, any block-local analysis would fail to discover the equality $x_2 = y_2$.

CompCert is a realistic formally verified compiler that generates PowerPC, ARM or x86 code from source programs written in a large subset of C. CompCert formalizes the operational semantics of dozen intermediate languages, and proves for each phase a semantics preservation theorem. Preservation theorems are expressed in terms of program behaviors, i.e. finite or infinite traces of external function calls (a.k.a. events) that are performed during the execution of the program, and claim that individual compilation phases preserve behaviors. A consequence of the theorems is that for any C program `p` that does not go wrong, and target program `tp` output by the successful compilation of `p` by the compiler `compcert_compiler`, the set of behaviors of `p` contains all behaviors of the target program `tp`. The formal theorem is:

```
Theorem compcert_compiler_correct: ∀ (p: C.program) (tp: Asm.program),
  (not_wrong_program p ∧ compcert_compiler p = OK tp) →
  (∀ beh, exec_asm_program tp beh → exec_C_program p beh).
```

This paper focuses on the CompCert middle-end where most of the existing optimisations are performed (currently: constant propagation, removal of redundant cast, tail call detection, local value numbering and a register allocation phase that includes copy propagation). These operate on a Register Transfer Language (RTL), whose syntax and semantics is given in Fig. 5.4. A RTL program is defined as a set of global variables, a set of functions, and an entry node. Functions are modelled as records that include a function signature `fn_sig`, a CFG `fn_code` of instructions over pseudo-registers. The CFG is not a basic-block graph: it partially maps each CFG node to a single instruction, and we stick to this important design choice of CompCert. As explained by Knoop *et al* [KKS98], it allows for simpler implementations of code manipulations and simplifies correctness proofs of analyses or transformations, without impacting too much their efficiency.

```
Inductive instr  :=
  | Inop (pc: node)
  | Iop (op: operation) (args: list reg) (res: reg) (pc: node)
  | Iload (chk:chunk) (addr:addressing) (args: list reg) (res: reg) (pc: node)
  | Istore (chk:chunk) (addr:addressing) (args:list reg) (src: reg) (pc: node)
  | Icall (sig: signature) (fn:ident) (args: list reg) (res: reg) (pc: node)
  | Icond (cond: condition) (args: list reg) (ifso ifnot: node)
  | Ireturn (or: option reg).

Inductive state :=
 | State (stack: list stackframe) (* call stack *)
         (f: function)             (* current function *)
         (sp: val)                 (* stack pointer *)
         (pc: node)                (* current program point *)
         (rs: regset)              (* register state *)
         (m: mem)                  (* memory state *)
 | Callstate (stack: list stackframe) (f: fundef) (args: list val) (m: mem)
 | Returnstate (stack: list stackframe) (v: val) (m: mem).

Inductive step: state → trace → state → Prop :=
 | ex_Inop: ∀ s f sp pc rs m pc',
     fn_code f pc = Some(Inop pc') →
     step (State s f sp pc rs m) ϵ (State s f sp pc' rs m)
 | ex_Iop: ∀ s f sp pc rs m pc' op args res v,
     fn_code f pc = Some(Iop op args res pc') →
     eval_operation sp op (rs##args) m = Some v →
     step (State s f sp pc rs m) ϵ (State s f sp pc' (rs#res←v) m)
 | ex_Iload: ∀ s f sp pc rs m pc' chk addr args res a v,
     fn_code f pc = Some(Iload chk addr args res pc') →
     eval_addressing sp addr (rs##args) = Some a →
     Mem.loadv chk m a = Some v →
     step (State s f sp pc rs m) ϵ (State s f sp pc' (rs#res←v) m)
```

Figure 5.4: Syntax and semantics of RTL (excerpt)

The RTL instruction set includes arithmetic operations (`Iop`), memory loads (`Iload`) and stores (`Istore`), function calls (`Icall`), conditional (`Icond`) and unconditional jumps (`Inop`), and a return statement (`Ireturn`)— for brevity, we do not discuss here jumptables and other kinds of function calls: call to a function pointer stored in a register, tail calls, and built-in functions. All instructions take as last argument a node `pc` denoting the next instruction to be executed; additionally, all instructions but `Inop` take as arguments pseudo-registers of type `reg`, memory chunks, and addressing modes.

The type of states is defined as the tagged union of regular states, call states and return states (Fig. 5.4). We focus on regular states, as we only expose here the intra-procedural part of the language. A regular semantic state (`State`) is a tuple that contains a call stack (representing the current pending function calls), the current function description and stack pointer (to the stack data block, a part of the global memory where variables dereferenced in the C source program reside), the current program point, the registers state (a mapping of local variables to values) and the global memory. The semantics also includes a global environment mapping function names and global variables to memory addresses; it is never modified during a program execution, and thus ommitted in our presentation.

The operational behavior of programs is modelled by the relation `step` between two semantic states (see Fig. 5.4), and a trace of events; all instructions except function calls do not emit any event, hence the transitions that they induced are tagged by the empty event trace ϵ. We briefly comment on the rules: (`Inop pc`$'$) branches to the next program point `pc`$'$. (`Iop op args res pc`$'$) performs the arithmetic operation `op` over the values of registers

args (written **rs##args**), stores the result in **res** (written **rs#res** ← **v**), and branches to **pc′**. The instruction (**Iload chk addr args res pc′**) loads a **chk** memory quantity from the address determined by the addressing mode **addr** and the values of the **args** registers, stores the quantity just read into **res**, and branches to **pc′**.

5.2 The SSA language

We describe the syntax and operational semantics of the language SSA that provides the SSA form of RTL programs. We equip the notion of SSA program with a *well-formedness* predicate capturing essential properties of SSA forms.

SSA programs Our definition of SSA program distinguishes between RTL-like instructions and ϕ-functions; the distinction avoids the need for unwieldy mappings between program points when converting to SSA, and allows for a smooth integration in CompCert. Fig. 5.5 introduces the syntax of SSA. SSA functions operate on indexed registers of type **SSA.reg** = **RTL.reg** ∗ **idx**, and include an additional field **fn_phicode** mapping junction points to ϕ-blocks. The latter are modelled as lists of ϕ-functions of the form (**Iphi args res**), where **res** is an indexed register, and **args** a list of indexed registers.

Next, we define structural constraints that allow giving an intuitive semantics to SSA programs. First, we require that the domain of the function **fn_phicode** be the set of junction points. Second, we require that all ϕ-functions in a ϕ-block have the same number of arguments as the number of predecessors of that block. Third, we require that all predecessors of a junction point be (**Inop pc**) instructions. This is a mild constraint, that can be ensured systematically on RTL programs through normalization, and that will carry over to their SSA forms. Fig. 5.6 shows the RTL program from Fig. 5.2 after normalization.

Finally, we consider two essential properties of SSA forms: unique definitions and strictness. The unique definitions property states that each register is uniquely defined, whereas the strictness property states that each variable use is dominated by the (unique) definition of that variable. While the two properties are closely related, none implies the other; the program $[y_0 := x_0; x_0 := 1]$ satisfies the unique definitions property but is not in strict form whereas the program $[x_0 := 1; x_0 := 2; y_0 := x_0]$ is strict but does not satisfy the unique definitions property. To formalize these properties, one first defines the type of paths in a CFG, and predicates **dom** and **sdom** for dominance and strict dominance. Then, one must define the two predicates **def**, **use** of type **SSA.function** → **SSA.reg** → **node** → **Prop** such that proposition **def f x pc** (respectively **use f x pc**) holds iff the register **x** is defined (resp. used) at node **pc** in the (RTL-like or ϕ-) code of the function **f**. The definition of **use** is complex because variables may be used in ϕ-functions: the widely adopted convention is to view ϕ-functions as lazily evaluated, their ith argument thus being used at the ith predecessor of the instruction. For example, in the SSA program of Fig. 5.6, variable $\mathtt{x}_2$ is defined at node 6 and used at node 8, the 2nd predecessor of the junction point 9 where $\mathtt{x}_2$ appears as 2nd argument of the ϕ-function. A use in the regular code is more straightforward: a variable is used by an instruction if it appears on its right-hand side. Using **def** and **use**, one can then state the unique definition and strictness properties, and well-formedness. Formally, we say that a SSA function is well-formed if it satisfies the following predicate:

```
Record wf_ssa_function (f:SSA.function) : Prop := {
  fn_ssa:          unique_def f;
  fn_strict:       ∀ x u d, use f x u → def f x d → dom f d u;
```

```
Inductive instr := ...

Inductive phiinstr :=
 | Iphi (args: list SSA.reg)
        (res: SSA.reg).

Definition phiblock:= list phiinstr.
```

```
Record function := {
 fn_sig: signature;          signature
 fn_params: list SSA.reg;    parameters
 fn_stacksize: Z;   activation record size
 fn_code: code;              code graph
 fn_phicode: phicode;        ϕ-blocks graph
 fn_entrypoint: node}.       entry node
```

```
Inductive step: SSA.state → trace → SSA.state → Prop :=
  | ex_Inop_njp: ∀ s f sp pc rs m pc',
     fn_code f pc = Some(Inop pc') →
     ¬ join_point pc' f →
     step (State s f sp pc rs m) ϵ (State s f sp pc' rs m)
  | ex_Inop_jp: ∀ s f sp pc rs m pc' phib k,
     fn_code f pc = Some(Inop pc') →
     join_point pc' f →
     fn_phicode f pc' = Some phib →
     index_pred f pc pc' = Some k →
     step (State s f sp pc rs m) ϵ (State s f sp pc' (phistore k rs phib) m)

Fixpoint phistore k rs phib : nat → SSA.regset → phiblock → SSA.regset :=
  match phib with
    | nil => rs
    | (Iphi args res)::phib =>
      match nth_error args k with
        | None => rs
        | Some arg => (phistore k rs phib)#res ← (rs#arg)
      end end.
```

Figure 5.5: Syntax and semantics of SSA (excerpt)

```
 fn_wf_block:    block_nb_args f;
 fn_block_at_jp: ∀ jp, join_point jp f ↔ fn_phicode f jp ≠ None;
 fn_normalized:  ∀ jp pc, join_point jp f → In jp (succs f pc) →
                          fn_code f pc = Some (Inop jp);}.
```

where predicates `unique_def` and `block_nb_args` respectively capture that a function satisfies the unique definitions property and the structural constraint about arguments. In the sequel, we show that conversion to SSA yields well-formed programs. Besides, our SSA-based optimizations will assume that the input SSA programs are well-formed; in turn, we prove for each of them that output programs are well-formed.

Semantics The notion of SSA state is similar to the notion of RTL state, except that the type of registers and current function are modified into `SSA.reg` and `SSA.function` respectively. The small-step operational semantics is defined on SSA programs that satisfy the structural constraints introduced in the previous paragraph. Formally, we define `SSA.step` as a relation between pairs of (SSA) states and a trace of events. The definition follows the one of `RTL.step`, except for instructions of the form (`Inop pc'`), where one distinguishes whether `pc'` is a junction point or not. In the latter case, the semantics coincide with the RTL semantics, i.e. the program point is updated in the semantic state. If on the contrary `pc'` is a junction point, then one executes the ϕ-block attached to `pc'` before the control flows to `pc'`. Executing ϕ-blocks on the way to `pc'` avoids the need to instrument the semantics of SSA with the predecessor program point, and crisply captures the intuitive meaning given to ϕ-blocks by Cytron *et al* (see Section 5.1). Note in particular that the normalization ensures the predecessor of a junction point is an `Inop` instruction. This greatly simplifies the definition of the semantics, and subsequently the proofs about SSA programs.

Following conventional practice, ϕ-blocks are given a parallel (big-step) semantics. This is formally embedded in the rule for `phistore` (Fig. 5.5). When reaching a join point `pc'` from its `k`th predecessor, we update the register set `rs` for each register `res` assigned in the ϕ-block `phib` with the value of register `arg` in `rs` (written `rs#arg`), where `arg` is the `k`th operand in the ϕ-function of `res` (written `nth_error args k = Some arg`). With the same notations, `phistore` satisfies, on well-formed SSA functions, a *parallel assignment* property:

```
∀ arg res, In (Iphi args res) phib →
   nth_error args k = Some arg → (phistore k rs phib)#res = rs#arg
```

5.3 Translation validation of SSA generation

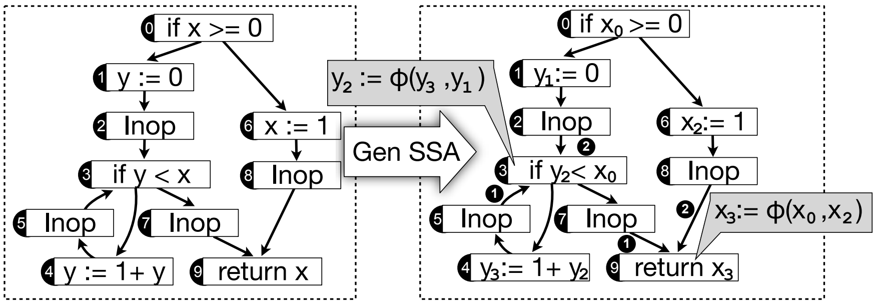
Modern compilers typically follow the algorithm by Cytron *et al* [CFR+91] to generate a minimal SSA form of programs in almost linear time w.r.t. the size of the program. The algorithm proceeds in four steps: (i) it computes the dominator tree of the CFG using the Lengauer and Tarjan algorithm [LT79]; (ii) it builds the dominance frontier using a bottom-up traversal of the dominator tree; (iii) for each variable, it places ϕ-functions using iterated dominance frontier; (iv) at last, it uses a top-down traversal of the dominator tree to rename each def and use of RTL variables with correct indexes. Programming efficiently the algorithm in Coq and proving formally its correctness is a significant challenge—even verifying formally Step (i) requires to formalize a substantial amount of graph theory. Instead, we provide a new validation algorithm that checks in linear time that a SSA program is a correct SSA form of an input RTL program. The algorithm is complete w.r.t. minimal SSA form, and can be enhanced by a liveness analysis to handle pruned and semi-pruned SSA forms. In order to be used in a certified compiler chain, we also show that our validator preserves behaviors.

Translation validation of SSA conversion is performed in two passes. The first pass performs a structural verification on programs: given a RTL function `f` and a SSA function `tf`, it verifies that `tf` satisfies all clauses of well-formedness except strictness, and that the code of `f` can be recovered from its SSA form `tf` simply by erasing ϕ-blocks and variable indices—the latter property is captured formally by the proposition `structural_spec f tf`. The second pass relies on a type system to ensure strictness and semantics-preservation. Overall the pseudo-code of the validator is:

```
let SSA_validator (f: RTL.function) (tf: SSA.function): bool :=
  if     (check_blocks_are_wf tf)          (* ensures block_are_wf tf *)
     && (check_blocks_are_at_jp tf)        (* ensures block_at_jp tf *)
     && (check_normalized tf)              (* ensures normalization *)
     && (check_unique_def tf)              (* ensures unique_def tf *)
     && (check_structural_spec f tf)       (* ensures structural_spec f tf *)
  then (is_well_typed f tf) else false
```

where `is_well_typed f tf` is the predicate stating that the function is well-typed w.r.t. our type system for SSA form.

Type system The basic idea of our type system is to track for each variable its *last* definition; this is achieved by assigning to all program points a local typing, i.e., an element of `ltype` = `RTL.reg` → `idx`; we let γ range over local typings. Then, the global typing of an SSA function `tf` is an element of `gtype` = `node` → `RTL.reg` → `idx`; we let Γ range over global typings. The type system performs various verifications between SSA and RTL registers. For



pc	(Γ pc x)	(Γ pc y)	(live pc)
0	0	0	{x}
1	0	0	{x}
2	0	1	{x, y}
3	0	2	{x, y}
4	0	2	{x, y}
5	0	3	{x, y}
6	0	0	{ }
7	0	2	{x}
8	2	0	{x}
9	3	0	{x}

Figure 5.6: A RTL program and its pruned SSA form and the corresponding type information

example, it verifies that if at node `pc`, an instruction uses the variable (`x`, `i`) then (Γ `pc x` = `i`). Figure 5.6 provides the intermediate types for our running example. We refer to [BDP12] for the detail of the typing rules.

Liveness As explained in Section 5.1, liveness information can be used to minimize the number of ϕ-functions in a SSA program; specifically, ϕ-blocks only need to assign live variables (in Fig. 5.6, the variable y is live at node 3, and x is live at node 9). Hence, our type system is parametrized by a function `live` that models a liveness analysis. Formally, we require that the `live` function satisfy two properties (for a function `f`, their conjunction is denoted by (`wf_live f live`)): (i) if a variable is used at a program point, then it should be live at this point and (ii) a variable that is live at a given program point is, at the predecessor point, either live or assigned.

Our type system is able to handle different SSA forms through appropriate instantiations of `live`. Our formalization provides support for minimal SSA and pruned SSA forms, respectively by defining `live` respectively as the trivial over-approximation (for each point, it is the set of all the RTL variables), and the result of a standard liveness analysis. One could also support semi-pruned forms, by instantiating `live` as the result of the block-local liveness analysis of [BCHS98].

Our Coq implementation has been highly engineered in order to speed up the validation. In particular, it performs type inference rather than type checking; the algorithm performs a single, linear scan of the program, and checks the list of arguments of ϕ-functions only once per junction point, rather than once per incoming edge for a given join point. On the benchmarks given in Section 5.6, our efficient implementation is ten times faster than a naive type checker derived from the non-executable type system.

Properties of the validator

Strictness All SSA programs accepted by the type system are strict. It follows that only well-formed SSA functions will be accepted by the validator.

```
Theorem wt_strict: ∀ f tf Γ live,
wf_live f live → wt_function tf Γ live →
∀ (xi : SSA.reg) (u d : node), use tf xi u → def tf xi d → dom tf d u.
```

The proof of `wt_strict` relies on two auxiliary lemmas about local typings in well-typed functions. The first lemma states that if a variable (`x`, `i`) is used at node `pc`, then it must be that (Γ pc x = i). The second lemma states that whenever (Γ pc x = i), the definition point of variable (x, i) dominates pc.

Soundness The validator is sound in the sense that if it accepts a RTL program f and an SSA form tf, then all behaviors of tf are also behaviors of f. Since CompCert already shows the general result that a lock-step forward simulation implies preservation of behaviors, it is sufficient to exhibit such a simulation:

```
Theorem validator_correct : ∀ (prog:RTL.program) (tprog:SSA.program),
SSA_validator prog tprog = true →
 ∀ s1 t s2, RTL.step s1 t s2 →
   ∀ s1', s1 ≃ s1' → ∃ s2', SSA.step s1' t s2' ∧ s2 ≃ s2'.
```

where the binary relation $\simeq$ between semantic states of RTL and SSA carries the invariants needed for proving behavior preservation. For instance, two regular states are related by $\simeq$ if their memory states, stack pointers, and program counters are equal, their function descriptions are suitably related, e.g. by structural_spec, and their register states rs and rs′ agree, i.e. satisfy (agree (Γ pc) rs rs′ (live pc)), where

```
Definition agree (γ:ltype) (rs:RTL.regset) (rs':SSA.regset) (live:Regset.t):=
  ∀ r, r ∈ live → rs#r  = rs'#(r, γ r).
```

Agreement is at the heart of the proof. It captures the semantics of local typings by making explicit how, at a given program point, variables of f should be interpreted in terms of the new variables in tf. The definition of $\simeq$ is completed by defining equivalence of stackframes; this relation basically lifts to the callstack all the invariants enforced by $\simeq$ (see [BDP] for the formal definition of $\simeq$).

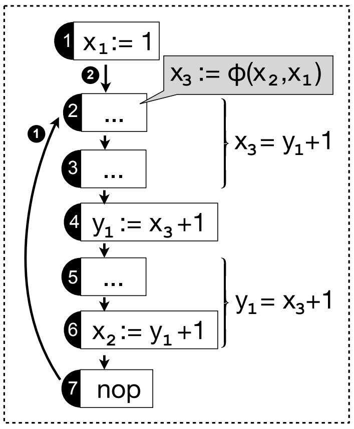
Completeness An essential property of our type system is that it accepts all the SSA programs that are output by the algorithm by Cytron *et al* [CFR+91]. The idea of the proof is as follows (provided in [BDP]). First, one defines for each RTL normalized program f a global typing Γ. Second, we show that all instructions of the program tf output by our implementation are typable. Then, we show that all edges are typable if we omit the constraints about correct use; the proof relies crucially on the fixpoint characterization of the iterated dominance frontier, as given in work of Cytron *et al* [CFR+91]. Finally, one shows that all constraints about correct use are satisfied, and hence the program tf is typable with Γ.

5.4 Conversion out of SSA

We have programed and verified a simple de-SSA algorithm that transforms SSA programs into RTL programs—so that they can be further processed by CompCert back end. The idea is to substitute each ϕ-function with one variable copy at each predecessor of junction points. Thanks to the single-instruction graph of RTL, replacing ϕ-functions with copies ensures soundness of the transformation, since critical edges are automatically splitted by code insertion—a critical edge is an edge whose entry has several successors and exit has several predecessors (see [BCHS98]). Pleasingly, the representation of programs inherited from CompCert deflates the penalty cost of splitting edges—on the contrary, algorithms that operate on *basic-block graphs* carefully avoid edge splitting, at the cost of making de-SSA algorithms significantly more complex. On the negative side, our current implementation of de-SSA fails on SSA programs with non-parallel ϕ-blocks, i.e. in which some variable is both used and defined. Minor future work includes making de-SSA total, reusing the formalization of the parallel moves algorithm [RSL08]—which transforms a set of parallel moves into an equivalent sequence of elementary moves (using additional temporaries), and that is already used in CompCert. Concerning the correctness of the transformation, we proceed by giving a forward simulation between the SSA program and the RTL program after de-SSA. The simulation requires the RTL program to perform several steps to simulate a (big-step) execution of a ϕ-block by the initial SSA program.

5.5 Validation of SSA-based optimizations

In this section, we introduce the *equation lemma* that supports the view of programs in SSA form as *systems of equations*. We then illustrate how to reason about a simple SSA-based optimization, namely copy propagation. Finally, we formalize and prove correct a GVN optimization.



Equation lemma The SSA representation provides an intuitive reading of programs: one can view the unique definition of a variable as an equation, and by extension one can view SSA programs as systems of equations. For instance, the definitions of `x3` and `y1` respectively induce the two equations $x_3 = y_1 + 1$ and $y_1 = x_3 + 1$. There is however a pitfall: the two equations entail $x_3 = x_3 + 2$, and thus are inconsistent. In fact, equations are only valid at program nodes dominated by the definition that induce them, as captured formally by the *equation-lemma* of SSA:

```
Lemma equation_lemma : ∀ prog d op args x succ f m rs sp pc s,
  wf_ssa_program prog →
    reachable prog (State s f sp pc rs m) →
    fn_code f d  = Some (Iop op args x succ) →
    sdom f d pc →
    eval_operation sp op (rs##args) m = Some (rs#x).
```

where `reachable` is a predicate that defines reachable states. In practice, it is often convenient to rely on a corollary that proves the validity of the defining equation of `x` at program points where `x` is used – thus avoiding reasoning on the dominance relation. The formal statement of the corollary is obtained by replacing the hypothesis `sdom f d pc` by the hypothesis `use f x pc`; the proof of the corollary intensively uses the strictness property of well-formed SSA programs.

We conclude with a succinct account of applying the corollary to prove the soundness of copy propagation (CP)—recall that CP will search for copies `x` := `y` and replace every use of `x` by a use of `y`. Suppose `pc` is a program point where such a replacement has been done. Every time `pc` is reached during the program execution, we are able to derive, using the corollary, that `rs`#`y` = `rs`#`x`, where `rs` is the current register state because (i) `y` is the right hand side of the definition of `x` and (ii) `pc` was a use point of `x` in the initial program. On non-SSA forms, the reasoning is more involved since one has to prove that the reaching definition for `x` is unique at `pc`, and that no redefinition of `y` can occur in between.

Global Value Numbering Our implementation of GVN is made of two components. The first one is an efficient but untrusted analysis, written in OCaml, for computing numberings

```
Inductive ≡^𝒩 : reg → reg → Prop :=
| GVN_refl : ∀ x, ≡^𝒩 x x
| GVN_Iop : ∀ x y pc1 pc2 op args1 args2 pc1' pc2'
    fn_code f pc1 = Some(Iop op args1 x pc1') → same_number 𝒩 args1 args2 →
    fn_code f pc2 = Some(Iop op args2 y pc2') → ≡^𝒩 x y
| GVN_Phi : ∀ x y pc args_x args_y
    fn_phicode f pc = Some phib → same_number 𝒩 args_x args_y →
    (Iphi args_x x) ∈ phib → (Iphi args_y y) ∈ phib → ≡^𝒩 x y.

Definition GVN_spec (𝒩:reg → reg) : Prop :=
(∀ x y, 𝒩 x = 𝒩 y → param f x → param f y→ x=y)∧(∀ x y, 𝒩 x = 𝒩 y → ≡^𝒩 x y).
```

Figure 5.7: Valid numbering

of SSA programs. From an abstract interpretation point of view, the analysis—which follows [AWZ88]—computes a fixpoint in the abstract domain of congruence partitions, where partitions are modelled as mappings $\mathcal{N}$: `reg` → `reg` that map a register to the canonical register of its equivalence class, and ordered w.r.t. reverse inclusion of equivalence kernels—recall that the equivalence kernel of $\mathcal{N}$ is the relation ∼ defined by `x` ∼ `y` if and only if $\mathcal{N}\ x = \mathcal{N}\ y$. Viewing the result of the analysis as a post-fixpoint is the key to our second component, a validator that checks whether a numbering $\mathcal{N}$ is indeed a post-fixpoint of the analysis on a program `p`, and if so returns an optimized SSA program `tp`. The validator is programmed in Coq, and is accompanied with a proof that optimized programs preserve the behaviors of the original programs.

The notion of valid numbering is formally defined in Fig. 5.7. First, we define for each numbering $\mathcal{N}$ the relation $\equiv^{\mathcal{N}}$ as the smallest reflexive relation identifying: (i) registers whose assignments share the same operator and corresponding arguments are equivalent w.r.t. $\mathcal{N}$ (predicate `same_number`); (ii) registers that are defined in the same ϕ-block with equivalent arguments.Then, for a numbering $\mathcal{N}$ to be valid (see `GVN_spec`), its equivalence kernel must not contain a pair of distinct function parameters and it must moreover be included in $\equiv^{\mathcal{N}}$. The latter ensures the intended post-fixpoint property.

The crux of the correctness proof of the GVN validator is the correctness lemma for a valid numbering: if $\mathcal{N}$ is a valid numbering for `f`, and `rs` is a register state that can be reached at node `pc`, and `x` and `y` are two registers whose definition strictly dominate `pc`, then $\mathcal{N}$ `x` = $\mathcal{N}$ `y` entails that `rs` holds equal values for `x` and `y`:

```
Lemma valid_numbering_correct : ∀ prog s sp pc rs m,
    wf_ssa_program prog → GVN_spec 𝒩 →
      reachable prog (State s f sp pc rs m) → gamma 𝒩 pc rs.
```

where `gamma` is defined by

```
Definition gamma (𝒩:reg → reg) (pc:node) (rs: regset) : Prop :=
  ∀ x y: reg, def_sdom f x pc → def_sdom f y pc → 𝒩 x = 𝒩 y → rs#x = rs#y.
```

and `def_sdom f x pc` states that the definition of `x` in `f` strictly dominates `pc`. Let us illustrate this property with Fig. 5.3; registers $\mathtt{x}_2$ and $\mathtt{y}_2$ share the same numbering; they are indeed equal just after the assignment of $\mathtt{y}_2$ but not before.

Next, we describe the Coq implementation for optimizing SSA programs. The implementation takes as input a numbering $\mathcal{N}$, and a partial mapping `crep` that takes as input a register `x` and node `pc` and returns, if it exists, a register `y` such that `x` and `y` are related by the equivalence kernel of $\mathcal{N}$, and the definition of `y` strictly dominates `pc`. For efficiency reasons, we do not check the correctness of `crep` a priori, but lazily during the construction of the optimized program. The optimizer proceeds as follows: first, it checks whether $\mathcal{N}$ satisfies the predicate `GVN_spec`. Then, for each assignment (`Iop op args x pc`) of the original SSA program, the optimizer checks whether `crep` provides a canonical representative `y` for `x` at node `pc`. If so, it checks whether the definition of `y` strictly dominates `pc`; this is achieved by means of a dominance analysis, computed directly inside Coq with a standard dataflow framework *a la* Kildall. Provided `y` is validated, we can safely replace the previous instruction by a move from `y` to `x`.

We conclude by commenting briefly on the soundness proof of the transformation. It follows a standard forward simulation proof where the correctness of the numbering is proved at the same time as the simulation itself. Noticeably, the CFG normalization turned out to be extremely valuable for this proof. Indeed, consider a step from node `pc` to node `pc'`: we have to prove that (`gamma` $\mathcal{N}$ `pc'` `rs`) holds, asumming (`gamma` $\mathcal{N}$ `pc rs`). We reason by case analysis: if the instruction at `pc` is not an `Inop` instruction, we know by normalization that `pc'` is not a junction point. In this case, (`def_sdom f x pc'`) is equivalent to (`def_sdom f x pc`) $\vee$ (`def f x pc`) which is particularly useful to exploit the hypothesis that (`gamma` $\mathcal{N}$ `pc rs`) holds.

5.6 Implementation and experimental results

We have plugged in Compcert 1.8.2 our SSA middle-end made of (i) a Coq normalization (ii) an Ocaml SSA generator and its Coq validator; (iii) an Ocaml GVN inference tool and its Coq validator; (iv) a Coq de-SSA transformation. Our formal development adds 15.000 lines of Coq code and 1.000 lines of Ocaml to the 80.000 lines of Coq and 1.000 lines of Ocaml provided in CompCert. It does not add any axioms to CompCert. We use the Coq extraction mechanism to obtain a SSA-based certified compiler, that we evaluate experimentally using the CompCert benchmarks. These include around 75.000 lines of C code, and fall into three categories of programs (from 20 to 5.000 LoC): small computation kernels, a raytracer, and the theorem prover Spass[1]. Below we briefly comment on three key points: efficiency of the SSA validator; effectiveness of the GVN optimizer; efficiency of generated code.

SSA validator In order to be practical, validators must be more efficient than state-of-the-art implementations of the transformations that they validate. At first sight, this criterion may seem too demanding for SSA, since generation into SSA form is performed in almost linear time. However, experimental results are surprisingly good: overall converting a program into SSA form takes approximately twice longer than type-checking the output program. In more detail, the times for SSA generation—specialized to pruned SSA—distribute as follows: (i) 9% for normalization of RTL; (ii) 37% for liveness analysis of RTL (the liveness analysis is provided in the CompCert distribution); (iii) 35% for conversion to SSA using the untrusted OCaml implementation (based on state-of-the-art algorithms); (iv) 19% for validation using the verified validator. This distribution appears to be uniform on all benchmarks except on the biggest functions where the liveness analysis exhibits a non-linear complexity.

GVN optimizer We measure the effectiveness of our GVN analyzer by performing a GVN-based CSE right after (Local Value Numbering) LVN-based CSE implemented by CompCert,

[1]Spass is the largest (69.073 LoC), we only use it to evaluate the compilation time.

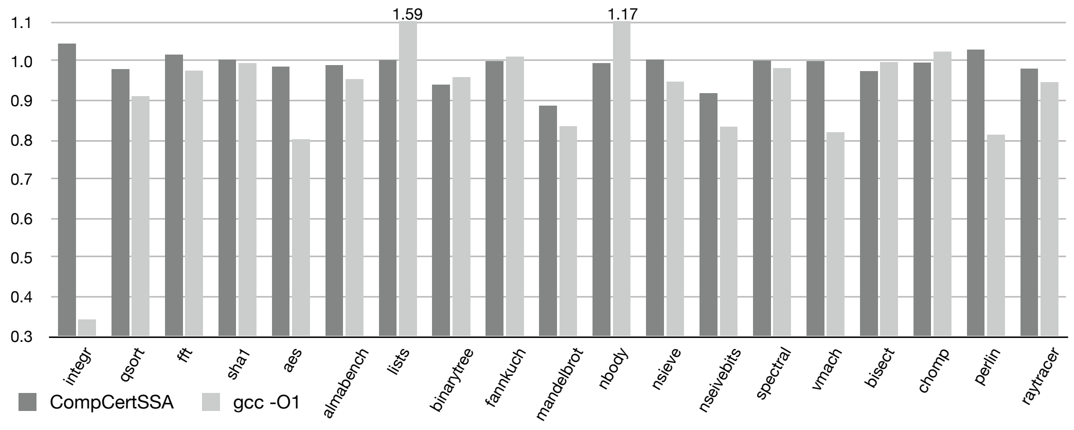


Figure 5.8: Execution times of generated code

counting how many additional `Iop` instructions are optimized by this additional CSE phase. To keep the comparison fair, we allow CompCert CSE to optimize around function calls—this is disabled in CompCert to keep the register pressure low. The overall improvement is significative: our global CSE optimizes an additional 25% of `Iop`.

Generated code To assess the efficiency of the generated code, we have compiled the benchmarks with three compilers: CompCert, our version of CompCert extended with a SSA middle-end (CompCertSSA), and `gcc −O1`. Fig. 5.8 gives the execution times *relative* to Compcert (shorter bars mean faster) on PowerPC. The test suite is too small to draw definite conclusions, but the results are encouraging. Our version of CompCert performs slightly better than CompCert. We expect that performance improves significantly by enhancing our middle-end with additional optimizations, and by relying on an SSA-based register allocator.

5.7 Related Work

Machine-checked formalizations Blech *et al* [BGLM05] use the Isabelle/HOL proof assistant to verify the generation of machine code from a representation of SSA programs that relies on term graphs. While graph-based representations may be useful for the untrusted parts of our compiler, they increase the complexity of the formal SSA semantics, and make it a greater challenge to verify SSA-based optimizations. They do not provide an algorithm to convert into SSA form, and leave as future work proving the correctness of SSA-based optimizations. Mansky and Gunter [MG10] use Isabelle/HOL to formalize and verify the conversion of CFG programs into SSA form. However, their transformation may yield non-minimal SSA, and does not aim extraction into efficient code. Moreover, it is not clear whether their semantics of SSA can be used to reason about optimizations. Zhao *et al* [ZNMZ12] formalize the LLVM intermediate representation in Coq. They define and relate several formal semantics of LLVM, including a static and dynamic semantics. They show how simple code motions can be validated with a simulation relation based on symbolic evaluation, and plan to extend the method to other transformations such as dead code elimination or constant propagation. Finally, there are several machine-checked accounts of Continuation Passing Style translations, e.g. [DL07], closely related to conversion to SSA form [App98].

Translation validation and type systems Menon *et al* [MGM$^+$06] propose a type system that can be used to verify memory safety of programs in SSA form, but their system does not enforce the SSA property. Matsuno and Ohori [MO06] define a type system equivalent to SSA: every typable program is given a type annotation makeing explicit def-use relations. Their type system is similar to ours except they type check one program w.r.t. annotations while we type check a pair of a RTL and a SSA program. They show that common optimizations such as dead code elimination and CSE are type-preserving. But they do not prove the semantics preservation of the optimizations. Stepp *et al* [STL11] report on a translation validator for LLVM. Their validator uses Equality Saturation [TSTL09], which views optimizations as equality analyses. Their tool does not validate GVN. Tristan *et al* [TGM11] independently report on an a translation validator for LLVM's inter-procedural optimizations. This tool supports GVN, but is currently not certified.

5.8 Conclusions and Perspectives

Our work shows that verified and realistic compilers can rely on a SSA-based middle-end that implements state-of-the-art algorithms, and opens the way for a new generation of verified compilers based on SSA. Our Coq formalization and proofs, as well a technical report can be found at

`http://www.irisa.fr/celtique/ext/compcertSSA`

A priority for further work is to achieve a tighter integration of our middle-end into CompCert. There are three immediate objectives: (i) enhancing our SSA middle-end to handle memory aliases as done by CompCert RTL-based middle-end, (ii) implementing a SSA-based register allocator [HGG06], and (iii) verifying more SSA-based optimizations, including lazy code motion [KRS92]—we expect that our implementation of GVN will provide significant leverage there. Eventually, it should be possible to shift all CompCert optimisations into the SSA middle-end. In the longer term, it would be appealing to apply our methods to LLVM, building on [TGM11, STL11, ZNMZ12].

The primary goal of this work was a formal and mechanized study of the SSA representation. Our ambition is to make our verified static analysis platform benefits from this work. Embedding this work in the CompCert project has been very valuable for us. We hence obtain a great end-to-end story for this SSA formalisation.

Chapter 6

Conclusions and Perspectives

6.1 Summary

This document summarises several years of research for designing reliable static analyses. We base this research on the intensive use of the Coq proof assistant[1]. Such a tool allows to mechanically specify, program and prove correct static analysers with respect to a formal model of the programming language semantics.

We have tried to cover a large spectrum of static analysis in this document: specific abstract-interpretation-based numerical analyses are covered in Chapter 2, security analyses are presented in Chapter 3 (information flow) and Chapter 4 (object initialisation and secure object cloning). Concurrency is considered in Chapter 3 with a data race analysis. An advanced compiler optimisation is presented in Chapitre 5 with the global value numbering analysis on a SSA representation.

We believe that building such formal proofs and reasoning about large and non-trivial static analysers will help develop new "proof engineering" methods. Our initial motivation was to formally verified realistic static analysis tools for Java bytecode program. We have conducted large proofs in this area (Chapitre 3) but one of our conclusions was that better methodologies were required. To that aim, we provide a new static analysis platform, Sawja. The platform is not verified but is meant for a Coq embedding. We have specially based our implementation on functional data-structures (imperative programs are generally more difficult to prove correct than functional programs in current proof assistants) and have formalised a non-trivial program transformation from bytecode to an intermediate representation. We believe that such an intermediate representation (IR) is a keystone in a verified platform. Chapter 4 provides two advanced static analyses that are both verified on a simple object-oriented core language and implemented into a robust prototype using Sawja. We believe a good verified IR could provide an important bridge between both parts of the work. Still, modern compiler and static analyses rely on IRs with non-trivial semantic properties like the SSA representation. Chapter 5 provides a formal studied of this representation.

6.2 Methodologies

If we summarise the various methods we have employed in this work, two of them merit their own discussions.

On several occasions, we have introduced **intermediate semantics** in our developments. Starting from a program language and its canonical semantics, we introduce a second semantics before proving the correctness of the static analysis. This second layer is generally introduced

[1]Chapter 2, 3, 4 and 5 represent 75 Kloc of Coq.

in order to do a half step toward the final soundness proof. It is for example the case in Chapter 2 when we introduce a collecting semantics that mimics the fixpoint iteration strategy of the static analysis but keep the expressiveness of the standard operational semantics. The method is also illustrated in Chapter 3 for the verified points-to analysis. We introduce an instrumented semantics that manipulates directly the allocation context of each references. Such an information could be extracted from an execution history but our instrumentation simplifies many reasoning in the soundness proof and, at the same time, is provably harmless with respect to the studied notion of data races. Those *half* steps may look tiny from the perspective of a paper-proof but mechanised proofs often requires to decompose a step into several ones.

The second technique we have favored is *a posteriori* **validation**. In a static analyser, a compiler or any software system, some computation components may be very difficult to verify or/and to program efficiently in a fully functional language as Coq. Let us assume we would like to verify that a computation $F \in A \rightarrow B$ satisfies a specification $R \subseteq A \times B$. In a direct approach, we would (i) program F in the proof assistant (ii) and then prove that F is correct with respect to R:

$$\forall a \in A, (a, F(a)) \in R$$

With *a posteriori* validation, we do not need to program F in the proof assistant itself. Instead we program a validator $V \in A \times B \rightarrow$ boolean and each time a computation on a input a is required, we check the result $F(a)$ (given by an untrusted tool) with a call to $V(a, F(a))$. The validator must, upon success, provides the guarantee that F has built a correct output

$$\forall a \in A,\ \forall b \in B,\ V(a,b) = \text{true} \Rightarrow (a,b) \in R$$

We used this technique in Chapter 2. Instead of programming and proving correct the transfer function of a polyhedra abstract domain, we check each computation with a verified validator. In Chapter 5, we rely on an untrusted Ocaml implementation of Cytron *et al* algorithm [CFR+91] for computing minimal SSA form and validate each output with verified type checker. More generally, for any fixpoint computation of a static analysis we can let an untrusted tool perform the inference and check if the result is indeed a fixpoint.

6.3 Phd and Postdoc Supervision

This document is partially based on the work of two PhD students I have co-supervised with Thomas Jensen.

6.3.1 Foundations and Implementation of a Tool Bench for Static Analysis of Java Bytecode Programs

In his thesis, Laurent Hubert studies the static analysis of Java bytecode and its semantics foundations. The initialization of an information system is a delicate operation where security properties are enforced and invariants installed. Initialization of fields, objects and classes in Java are difficult operations. These difficulties may lead to security breaches and to bugs, and make the static verification of software more difficult. The thesis proposes static analyses to better master initialization in Java. Hence, Laurent proposes a null pointer analysis that finely tracks initialization of fields [HJP08a]. It allows proving the absence of dereferencing of null pointers (NullPointerException) and refining the intra-procedural control flow graph.

He presents another analysis to refine the inter-procedural control flow due to class initialization [HP09]. This analysis directly allows inferring more precise information about static fields. Finally, he proposes a type system that enforces secure object initialization, hence offering a sound and automatic solution to a known security issue [HJMP10]. These analyses have been proved correct in Coq using core idealized Java languages. We also provide implementations. We developed several tools from these analyses, with a strong focus at having sound but also efficient tools. To ease the adaptation of such analyses to the full-featured Java bytecode, we have developed the Sawja library. This work has been presented in Chapter 4.

This work has been conducted from September 2007 to December 2010. Laurent is now a research and development engineer working in a young french start-up, Prove&Run (leaded by Dominique Bolignano).

6.3.2 Semantic Foundations of Intermediate Program Representations

In her thesis, Delphine Demange, gives a formal, semantic account on intermediate representations (IRs), so that they can be leveraged in the formal proof of static analysers and compilers. She first studies a register-based IR of Java bytecode used in compilers and verifiers. She specifies the IR generation by a semantic theorem stating what the transformation preserves, e.g. object initialization or exceptions, but also what it modifies and how, e.g. object allocation [DJP10]. This IR is implemented in the Sawja platform (see Chapter 4). Then, she studies the Static Single Assignment (SSA) form, an IR widely used in modern compilers and verifiers. We have implemented and prove in Coq an SSA middle-end for the CompCert C compiler [BDP12]. Finally, she studies the semantics of concurrent Java IRs. Due to instruction reorderings performed by the compiler and the hardware, the current definition of the Java Memory Model (JMM) is complex, and unfortunately formally flawed. Targetting x86 architectures, we identify a subset of the JMM that is intuitive and tractable in formal proofs. We characterize the reorderings it allows, and factor out a proof common to the IRs of a compiler. This work is still in progress and has been initiated during an internship at Purdue University while I was in sabbatical leave in this University.

This thesis has been conducted from September 2009 to October 2012. Delphine will join Benjamin Pierce's research group at UPenn University for a postdoc position in November 2012.

6.4 Perspectives

Building a verified static analysis platform for Java bytecode programs is the next step we would like to perform. Our ambition is to follow as closely as possible the design of the Sawja library. Such an effort will certainly gives rise to new interesting research problems and eventually provide a new original platform for those who want to develop static analysers, event without formal proof at first aim.

One question that remains and seems particularly difficult to answer deals with multi-threaded Java programs. Indeed, their semantics is not so well understood. The Java Memory Model (JMM) provides a high level semantics for concurrent (and racy) programs, portable across architectures. But it appears that the official specification is flawed at subtle points. After several patches the main criticism against the JMM is his lack of formal connection with compiler techniques. The JMM is supposed to act as an interface between the programmer and the program transformations that compilers are performing behind the scene.

Building a verified platform requires a solid formalisation of the JMM. But the semantics itself needs some patches. I believe it will be hard to fix such a complex model without taking care of what these compilers are actually doing. Hence I propose not only to build a verified platform on top of the JMM but rather to make the JMM the middle of the verified toolchain. We should build below this model a verified compiler for the concurrent Java bytecode programming language[2]. In this area, the recent success of the CompCert C compiler has shown that it is possible to construct reasonably efficient compilers along with a machine-checked proof of their correctness. But performing the same effort for the Java platform would require to tackle new challenges because of the complexity of a JVM platform. Starting from state-of-the-art aggressive optimisations, we would like to better understand their formal semantic links with the JMM. Our aim is not to decide if a specific aggressive optimisation is useful or not in a Java compiler (some of these optimisations target very specific benchmarks). Instead, we would like to reconsider the design of these optimisations, understand which part can be *a posteriori* validated, which part can not and we would like to understand how they affect the relaxed memory model of Java. We believe that our past works on various advanced static analysis will help fulfil this objective.

This long term project is full of new challenging subjects. Among them we need to find a formal path from the JMM to an operational hardware model. In collaboration with Delphine Demange, Vincent Laporte and colleagues at Purdue University we are currently designing an intermediate memory model that could act as an intermediate step between these models. When a formal proof is required, one generally seek for an operational model of the language. Several attempts have been made to catch the operational view of the JMM but currently none is directly linked to the actual definition. We would like to consider a new form of memory model where legal executions are directly characterised in term of reordering. We have proved that such a model can be equivalent to a operational model for TSO memory models.

One other challenge is to build a verified compiler on top of hardware weak memory models. Weak memory models generally enjoy the so-called *data-race-free guarantee*: if a program has no data-races with respect to an interleaving semantics then the set of all its executions in the weak memory model should be equal to the set of its executions under an interleaving semantics. So, most of the intricate reasoning occur when data races are introduced by the compiler. We could ask ourselves if a compiler has a good reason to do that and indeed, yes! Modern Java compilers will do that. If the garbage collector they generate run on a distinct thread, it will regularly mark some memory cells as reachable while the user code is updating the same information. Those races are part of the design of such a tool.

To summarise, a verified concurrent Java bytecode compiler is full of challenges: JMM, concurrent racy codes generated by the compiler, Just-in-time compilation. These beasts are daily used on our computers with very few detected defects but they have never been formally linked together in a same formal proof. A better understanding of these techniques will certainly provide rich lessons for the design of new languages.

[2]By Java compiler we mean a compiler from bytecode to assembly code.

Bibliography

[AB07] A. W. Appel and S. Blazy. Separation logic for small-step Cminor. In *In Proc. of TPHOLs 2007*, volume 4732 of *LNCS*, pages 5–21. Springer-Verlag, 2007.

[ABF$^+$05] B. E. Aydemir, A. Bohannon, M. Fairbairn, J. Nathan Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses: The poplmark challenge. In *Proc. of TPHOLs 2005*, volume 3603 of *LNCS*, pages 50–65. Springer-Verlag, 2005.

[AFKT03] A. Aiken, J. S. Foster, J. Kodumal, and T. Terauchi. Checking and inferring local non-aliasing. In *Proc. of PLDI '03*, pages 129–140. ACM Press, 2003.

[AGN09] Z. Anderson, D. Gay, and M. Naik. Lightweight annotations for controlling sharing in concurrent data structures. In *Proc. of PLDI'09*, pages 98–109. ACM Press, 2009.

[App98] A.W. Appel. SSA is functional programming. *SIGPLAN Notices*, 33, 1998.

[App01] A. W. Appel. Foundational proof-carrying code. In *LICS 2001*, pages 247–258. IEEE, 2001.

[AWZ88] B. Alpern, M.N. Wegman, and F.K. Zadeck. Detecting equality of variables in programs. In *POPL'88*. ACM, 1988.

[BCDM02] G. Barthe, P. Courtieu, G. Dufay, and S. Melo de Sousa. Tool-Assisted Specification and Verification of the JavaCard Platform. In *Proc. of AMAST'02*, volume 2422 of *LNCS*, pages 41–59. Springer-Verlag, 2002.

[BCF$^+$99] M G. Burke, J. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M J. Serrano, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño dynamic optimizing compiler for java. In *Proc. of JAVA'99*, pages 129–141. ACM, 1999.

[BCHS98] P. Briggs, K.D. Cooper, T.J. Harvey, and L.T. Simpson. Practical improvements to the construction and destruction of static single assignment form. *SPE*, 1998.

[BCP11] F. Besson, P.E. Cornilleau, and D. Pichardie. Modular SMT proofs for fast reflexive checking inside coq. In *Proc. of CPP 2011*, volume 7086 of *LNCS*, pages 151–166. Springer-Verlag, 2011.

[BCS97] P. Briggs, K.D. Cooper, and L.T. Simpson. Value numbering. *SPE*, 1997.

[BDP] G. Barthe, D. Demange, and D. Pichardie. A formally verified SSA-based middle-end - Static Single Assignment meets CompCert - journal version in preparation. http://www.irisa.fr/celtique/ext/compcertSSA.

[BDP12] G. Barthe, D. Demange, and D. Pichardie. A formally verified SSA-based middle-end - Static Single Assignment meets CompCert. In *Proc. of ESOP 2012*, volume 7211 of *LNCS*, pages 47–66. Springer-Verlag, 2012.

[Ber09a] Y. Bertot. Structural abstract interpretation: A formal study using Coq. In *International LerNet ALFA Summer School 2008*, volume 5520 of *LNCS*, pages 153–194. Springer-Verlag, 2009.

[Ber09b] Y. Bertot. Theorem proving support in programming language semantics. In *From Semantics to Computer Science — Essays in Honour of Gilles Kahn*, pages 337–362. Cambridge University Press, 2009.

[Bes06] F. Besson. Fast reflexive arithmetic tactics: the linear case and beyond. In *Types for Proofs and Programs*, pages 48–62. Springer LNCS vol. 4502, 2006.

[BGL06] Y. Bertot, B. Grégoire, and X. Leroy. A structured approach to proving compiler optimizations based on dataflow analysis. In *Proc. of TYPES 2004*, volume 3839 of *LNCS*, pages 66–81. Springer-Verlag, 2006.

[BGLM05] J.O. Blech, S. Glesner, J. Leitner, and S. Mülling. Optimizing code generation from SSA form: A comparison between two formal correctness proofs in Isabelle/HOL. In *COCV'05*, ENTCS. Elsevier, 2005.

[BJPT10] F. Besson, T. Jensen, D. Pichardie, and T. Turpin. Certified result checking for polyhedral analysis of bytecode programs. In *Proc. of TGC 2010*, volume 6084 of *LNCS*, pages 253–267. Springer-Verlag, 2010.

[BKV09] N. Benton, A. Kennedy, and C. Varming. Some domain theory and denotational semantics in Coq. In *Proc. of TPHOLs '09*, volume 5674, pages 115–130. Springer-Verlag, 2009.

[Bla99] B. Blanchet. Escape analysis for object-oriented languages: Application to Java. In *Proc. of OOPSLA*, pages 20–34. ACM Press, 1999.

[BN05] A. Banerjee and D. Naumann. Stack-based access control for secure information flow. *Journal of Functional Programming*, 15:131–177, March 2005. Special Issue on Language-Based Security.

[BPR07] G. Barthe, D. Pichardie, and T. Rezk. A Certified Lightweight Non-Interference Java Bytecode Verifier. In *Proc. of ESOP'07*, volume 4421 of *LNCS*, pages 125–140. Springer-Verlag, 2007.

[BPR12] G. Barthe, D. Pichardie, and T. Rezk. A Certified Lightweight Non-Interference Java Bytecode Verifier. *Mathematical Structures in Computer Science (MSCS)*, 2012. To appear.

[BS96] D. F. Bacon and P. F. Sweeney. Fast static analysis of C++ virtual function calls. In *Proc. of OOPSLA'96*, pages 324–341, 1996.

[Buc06] A. Buckley. JSR 202: Java class file specification update, December 2006. `http://jcp.org/en/jsr/detail?id=202`.

[C08] B. Chetali and Q. H. Nguyen 0002. Industrial use of formal methods for a high-level security evaluation. In *FM*, volume 5014 of *Lecture Notes in Computer Science*, pages 198–213. Springer, 2008.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of POPL'77*, pages 238–252. ACM Press, 1977.

[CC92] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In *Proc. of PLILP'92*, pages 269–295. LNCS, 1992.

[CCF$^+$07] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Combination of abstractions in the ASTRÉE static analyzer. In *Proc. of ASIAN'06*. Springer LNCS vol. 4435, 2007.

[CER10a] The CERT Sun Microsystems secure coding standard for Java, February 2010. `https://www.securecoding.cert.org/confluence/display/java/`.

[CER10b] CERT. *The CERT Sun Microsystems Secure Coding Standard for Java*, 2010. `https://www.securecoding.cert.org`.

[CFR+91] R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, and F.K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM TOPLAS*, 1991.

[CGD06] S. Coupet-Grimal and W. Delobel. A uniform and certified approach for two static analyses. In *Proc. of TYPES 2004*, volume 3839 of *LNCS*, pages 115–137. Springer-Verlag, 2006.

[CGS+99] J.D. Choi, M. G., M. J. Serrano, V. C. Sreedhar, and S. P. Midkiff. Escape analysis for java. In *Proc. of OOPSLA*, pages 1–19. ACM Press, 1999.

[CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proc. of 5th ACM Symp. on Principles of Programming Languages (POPL'78)*, pages 84–97. ACM Press, 1978.

[Cho] Chord: A program analysis platform for java.

[CJPR05] D. Cachera, T. P. Jensen, D. Pichardie, and V. Rusu. Extracting a data flow analyser in constructive logic. *Theoretical Computer Science*, 342(1):56–78, 2005.

[Cou] P. Cousot, Visiting Professor at the MIT Aeronautics and Astronautics Department, Course 16.399: Abstract Interpretation.

[Cou99] P. Cousot. The calculational design of a generic abstract interpreter. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design*. NATO ASI Series F. IOS Press, Amsterdam, 1999.

[CP10] D. Cachera and D. Pichardie. A certified denotational abstract interpreter. In *Proc. of ITP-10*, volume 6172 of *LNCS*, pages 9–24. Springer-Verlag, 2010.

[Dav05] Pichardie David. *Interprétation abstraite en logique intuitionniste : extraction d'analyseurs Java certifiés*. PhD thesis, Université Rennes 1, 2005. In french.

[DFW96] D. Dean, E.W. Felten, and D.S. Wallach. Java security: From HotJava to Netscape and beyond. *IEEE Symposium on Security and Privacy*, pages 190–200, 1996.

[DJP09] D. Demange, T. Jensen, and D. Pichardie. A provably correct stackless intermediate representation for Java bytecode. Research Report 7021, INRIA, 2009. http://www.irisa.fr/celtique/demange/bir/rr7021-3.pdf.

[DJP10] D. Demange, T. Jensen, and D. Pichardie. A provably correct stackless intermediate representation for Java bytecode. In *Proc. of APLAS 2010*, volume 6461 of *LNCS*, pages 97–113. Springer-Verlag, 2010.

[DL07] Z. Dargaye and X. Leroy. Mechanized verification of CPS transformations. In *LPAR'07*, LNCS. Springer-Verlag, 2007.

[DMM99] C. Dubois and V. Ménissier-Morain. Certification of a type inference tool for ml: Damas-milner within coq. *J. Autom. Reasoning*, 23(3-4):319–346, 1999.

[DP09] F. Dabrowski and D. Pichardie. A certified data race analysis for a Java-like language. In *Proc. of TPHOLs'09*, volume 5674 of *LNCS*, pages 212–227. Springer-Verlag, 2009.

[DS04] Z. Deng and G. Smith. Lenient array operations for practical secure information flow. In *CSFW*, pages 115–124. IEEE Computer Society, 2004.

[FL03] M. Fähndrich and K. R. M. Leino. Declaring and checking non-null types in an object-oriented language. *ACM SIGPLAN Notices*, 38(11):302–312, November 2003.

[FM03] S. N. Freund and J. C. Mitchell. A type system for the Java bytecode language and verifier. *J. Autom. Reasoning*, 30(3-4):271–321, 2003.

[FM07] J.-C. Filliâtre and C. Marché. The why/krakatoa/caduceus platform for deductive program verification. In *Proc. of CAV 2007*, volume 4590 of *LNCS*, pages 173–177. Springer-Verlag, 2007.

[FX07] M. Fähndrich and S. Xia. Establishing object invariants with delayed types. In *Proc. of OOPSLA 2007*, pages 337–350. ACM, 2007.

[GJSB05] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification (3rd Edition)*. The Java. Addison Wesley, 3rd edition edition, 2005.

[GMR$^+$95] J. Guttman, L. Monk, J. Ramsdell, W. Farmer, and V. Swarup. The VLISP verified Scheme system. *Lisp and Symbolic Computation*, 8(1-2):33–110, 1995.

[Gor88] M. J. C. Gordon. Mechanizing programming logics in higher-order logic. In *Current Trends in Hardware Verfication and Automatic Theorem Proving*, pages 387–439. Springer, 1988.

[GZ99] G. Goos and W. Zimmermann. Verification of compilers. In *Correct System Design, Recent Insight and Advances*, volume 1710 of *LNCS*, pages 201–230. Springer-Verlag, 1999.

[HAN08] A. Hobor, A. W. Appel, and F. Zappa Nardelli. Oracle semantics for concurrent separation logic. In *Proc. ESOP 2008*, volume 4960, pages 353–367. Springer-Verlag, 2008.

[HBB$^+$10] L. Hubert, N. Barré, F. Besson, D. Demange, T. Jensen, V. Monfort, D. Pichardie, and T. Turpin. Sawja: Static Analysis Workshop for Java. In *Proc. of FoVeOOS 2010*, volume 6461 of *LNCS*, pages 92–106. Springer-Verlag, 2010.

[HGG06] S. Hack, D. Grund, and G. Goos. Register allocation for programs in SSA form. In *Proc. of CC 2006*, LNCS. Springer-Verlag, 2006.

[HJMP10] L. Hubert, T. Jensen, V. Monfort, and D. Pichardie. Enforcing secure object initialization in Java. In *Proc. of ESORICS 2010*, volume 6345 of *LNCS*, pages 101–115. Springer-Verlag, 2010.

[HJP08a] L. Hubert, T. Jensen, and D. Pichardie. Semantic foundations and inference of non-null annotations. In *Proc. of FMOODS'08*, volume 5051 of *LNCS*, pages 132–149. Springer-Verlag, 2008.

[HJP08b] Laurent Hubert, Thomas Jensen, and David Pichardie. Semantic foundations and inference of non-null annotations. In *Proc. of FMOODS*, volume 5051 of *LNCS*, pages 132–149. Springer Berlin, June 2008.

[HMM12] P. Herms, C. Marché, and B. Monate. A certified multi-prover verification condition generator. In *Proc. of VSTTE 2012*, volume 7152 of *LNCS*, pages 2–17. Springer-Verlag, 2012.

[HP07] M. Huisman and G. Petri. The Java memory model: a formal explanation. In *Proc. of VAMP 2007*, 2007.

[HP09] L. Hubert and D. Pichardie. Soundly handling static fields: Issues, semantics and analysis. *Proc. of BYTECODE'09*, 253(5):15–30, 2009.

[HS09] C. Hammer and G. Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *Int. J. Inf. Sec.*, 8(6):399–422, 2009.

[IBM] IBM. The T.J. Watson Libraries for Analysis (Wala). http://wala.sourceforge.net.

[JKP11] T. Jensen, F. Kirchner, and D. Pichardie. Secure the clones: Static enforcement of policies for secure object copying. In *Proc. of ESOP 2011*, volume 6602 of *LNCS*, pages 317–337. Springer-Verlag, 2011.

[JKP12] T. Jensen, F. Kirchner, and D. Pichardie. Secure the clones: Static enforcement of policies for secure object copying. *Logical Methods in Computer Science (LMCS)*, 8(2), 2012.

[JM09] B. Jeannet and A. Miné. Apron: A library of numerical abstract domains for static analysis. In *CAV*, pages 661–667, 2009.

[JPL12] J.-H. Jourdan, F. Pottier, and X. Leroy. Validating lr(1) parsers. In *Proc. of ESOP 2012*, volume 7211 of *LNCS*, pages 397–416. Springer-Verlag, 2012.

[KKS98] J. Knoop, D. Koschützkil, and B. Steffen. Basic-block graphs: Living dinosaurs? In *CC*, LNCS. Springer-Verlag, 1998.

[KN03] G. Klein and T. Nipkow. Verified bytecode verifiers. *Theor. Comput. Sci.*, 3(298):583–626, 2003.

[KN06] G. Klein and T. Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. *ACM Transactions on Programming Languages and Systems*, 28(4):619–695, 2006.

[KRS92] J. Knoop, O. Rüthing, and B. Steffen. Lazy code motion. In *PLDI'92*, 1992.

[Ler06] X. Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *Proc. of POPL'06*, pages 42–54. ACM Press, 2006.

[Ler09a] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[Ler09b] X. Leroy. A formally verified compiler back-end. *JAR*, 43(4), 2009.

[Ler10] X. Leroy. Mechanized semantics. In *Logics and languages for reliability and security*, volume 25 of *NATO Science for Peace and Security Series D: Information and Communication Security*, pages 195–224. IOS Press, 2010.

[LLV] The LLVM compiler infrastructure. http://llvm.org/.

[LMO07] P. Lammich and M. Müller-Olm. Formalization of conflict analysis of programs with procedures, thread creation, and monitors. In *The Archive of Formal Proofs*, 2007.

[Loc10] A. Lochbihler. Verifying a Compiler for Java Threads. In *ESOP*, pages 427–447, 2010.

[Loc12] A. Lochbihler. Java and the Java memory model – a unified, machine-checked formalisation. In *Proc. of ESOP*, 2012.

[LR12] X. Leroy and V. Robert. A formally-verified alias analysis. In *Proc. of CPP 2012*, LNCS. Springer-Verlag, 2012. To appear.

[LT79] T. Lengauer and R.E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM TOPLAS*, 1979.

[M. 88] M. J. C. Gordon. Mechanizing programming logics in higher-order logic. In G.M. Birtwistle and P.A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automatic Theorem Proving*, pages 387–439. Springer-Verlag, 1988.

[MG10] W. Mansky and E. Gunter. A framework for formal verification of compiler optimizations. In *ITP'10*. Springer-Verlag, 2010.

[MGM$^+$06] V. Menon, N. Glew, B.R. Murphy, A. McCreight, T. Shpeisman, A.R. Adl-Tabatabai, and L. Petersen. A verifiable SSA program representation for aggressive compiler optimization. In *POPL'06*. ACM, 2006.

[Min06a] A. Miné. Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In *Proc. of LCTES '06*, pages 54–63. ACM, 2006.

[Min06b] A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19:31–100, 2006.

[MO06] Y. Matsuno and A. Ohori. A type system equivalent to static single assignment. In *PPDP'06*. ACM, 2006.

[Moo96] J. S. Moore. *Piton: a mechanically verified assembly-language*. Kluwer, 1996.

[MP67] J. McCarthy and J. Painter. Correctness of a compiler for arithmetical expressions. In *Mathematical Aspects of Computer Science*, volume 19, pages 33–41. American Mathematical Society, 1967.

[MW72] R. Milner and R. Weyhrauch. Proving compiler correctness in a mechanized logic. In *Proc. 7th Annual Machine Intelligence Workshop*, volume 7 of *Machine Intelligence*, pages 51–72. Edinburgh University Press, 1972.

[Mye99] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. of POPL'99*, pages 228–241. ACM Press, 1999.

[Myr10] M. O. Myreen. Verified just-in-time compiler on x86. In *Proc. of POPL'10*, pages 107–118. ACM, 2010.

[NA07] M. Naik and A. Aiken. Conditional must not aliasing for static race detection. In *Proc. of POPL '07*, pages 327–338. ACM Press, 2007.

[Nai08] M. Naik. *Effective Static Data Race Detection For Java*. PhD thesis, Standford University, 2008.

[Nau05] D. A. Naumann. Verifying a secure information flow analyzer. In *Proc. of TPHOLs 2005*, volume 3603 of *LNCS*, pages 211–226. Springer-Verlag, 2005.

[NAW06] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *Proc. of PLDI '06*, pages 308–319. ACM Press, 2006.

[Nec00] G. Necula. Translation validation for an optimizing compiler. In *PLDI'00*, pages 83–94. ACM, 2000.

[Nip98] T. Nipkow. Winskel is (almost) right: Towards a mechanized semantics. *Formal Aspects of Computing*, 10(2):171–186, 1998.

[NN99] W Naraschewski and T. Nipkow. Type inference verified: Algorithm W in Isabelle/HOL. *J. Autom. Reasoning*, 23(3-4):299–318, 1999.

[Ora10] Oracle. *Secure Coding Guidelines for the Java Programming Language, version 3.0*, 2010. `http://java.sun.com/security/seccodeguide.html`.

[OYR04] P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *Proc. of POPL'04*, pages 268–280. ACM Press, 2004.

[P$^+$08] W. Paul et al. The Verisoft project. `http://www.verisoft.de/`, 2003–2008.

[PH08] G. Petri and M. Huisman. BicolanoMT: a formalization of multi-threaded Java at bytecode level. In *Proc. of Bytecode 2008*, ENTCS, 2008.

[PHN12] A. Popescu, J. Hölzl, and T. Nipkow. Proving concurrent noninterference. In *Proc. of CPP 2012*, LNCS. Springer-Verlag, 2012. To appear.

[Pic08] D. Pichardie. Building certified static analysers by modular construction of well-founded lattices. In *Proc. of FICS'08*, volume 212 of *ENTCS*, pages 225–239, 2008.

[PSS98] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *TACAS'98*, LNCS. Springer-Verlag, 1998.

[QM09] X. Qi and A. C. Myers. Masked types for sound object initialization. In *Proc. of POPL 2009*, pages 53–65. ACM, 2009.

[RSL08] L. Rideau, B.P. Serpette, and X. Leroy. Tilting at windmills with Coq: Formal verification of a compilation algorithm for parallel moves. *J. Autom. Reason.*, 40(4):307–326, 2008.

[SA98] R. Stata and M. Abadi. A type system for java bytecode subroutines. In *Proc of POPL,98*, pages 149–160. ACM Press, 1998.

[Sec03] Secunia Advisory SA10056: Sun JRE and SDK untrusted applet privilege escalation vulnerability. Web, October 2003. http://secunia.com/advisories/10056/.

[SM03] A. Sabelfeld and A. Myers. Language-based information-flow security. *J. Strategic Areas in Computing*, 21:5–19, 2003.

[SRW02] S. Sagiv, T. W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002.

[STL11] M. Stepp, R. Tate, and S. Lerner. Equality-based translation validator for LLVM. In *CAV'11*, LNCS. Springer-Verlag, 2011.

[Sun10] Sun. Secure coding guidelines for the Java programming language, version 3.0. Technical report, Oracle, 2010. http://java.sun.com/security/seccodeguide.html.

[TGM11] J.B. Tristan, P. Govereau, and G. Morrisett. Evaluating value-graph translation validation for LLVM. In *PLDI'11*. ACM, 2011.

[TL09] J.B. Tristan and X. Leroy. Verified validation of lazy code motion. In *PLDI'09*. ACM, 2009.

[TL10] J.B. Tristan and X. Leroy. A simple, verified validator for software pipelining. In *POPL'10*. ACM, 2010.

[TSTL09] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner. Equality saturation: a new approach to optimization. In *POPL'09*. ACM, 2009.

[TT97] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[UL08] C. Unkel and M. S. Lam. Automatic inference of stationary fields: a generalization of Java's final fields. In *Proc. of POPL 2008*, pages 183–195. ACM, 2008.

[VRCG$^+$99] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - A Java bytecode optimization framework. In *Proc. of CASCON*. IBM, 1999.

[VS97] D. Volpano and G. Smith. A type-based approach to program security. In *Proc. of TAPSOFT'97*, volume 1214 of *LNCS*, pages 607–621. Springer-Verlag, 1997.

[Was10] D. Wasserrab. *From Formal Semantics to Verified Slicing - A Modular Framework with Applications in Language Based Security*. PhD thesis, Karlsruher Institut für Technologie, Fakultät für Informatik, 2010.

[WCN05] M. Wildmoser, A. Chaieb, and T. Nipkow. Bytecode analysis for proof carrying code. In *Proc. of BYTECODE 2005*, ENTCS. Elsevier, 2005.

[WN05] M. Wildmoser and T. Nipkow. Asserting bytecode safety. In *Proc. of ESOP'05*, LNCS. Springer-Verlag, 2005.

[ZNMZ12] J. Zhao, S. Nagarakatte, M. M.K. Martin, and S. Zdancewic. Formalizing the LLVM intermediate representation for verified program transformations. In *Proc. of POPL'12*. ACM, 2012.