The SSA Representation Framework: Semantics, Analyses and GCC Implementation

Sebastian Pop

MINES PARIS

Collège doctoral

ÉD n^o 431 : Information, communication, modélisation et simulation

N^o attribué par la bibliothèque:

| | | | | | | | | | | |

THESE

pour obtenir le grade de

Docteur de l'Ecole des Mines de Paris

Spécialité "Automatique, Robotique, et Informatique Temps Réel"

présentée et soutenue publiquement par

Sebastian POP

le 13 Décembre 2006

La représentation SSA: sémantique, analyses et implémentation dans GCC

The SSA Representation Framework: Semantics, Analyses and GCC Implementation

Directeur de thèse : François Irigoin

Jury

Philippe	Clauss	Président
Albert	Cohen	Examinateur
Matthieu	Martel	Rapporteur
Lawrence	Rauchwerger	Rapporteur
Georges-André	Silber	Directeur

Contents

List of Figures

Chapter 1

Introduction

Programming languages appeared short after the first computing machines, and intended to abstract machines architectural specifics. The creation of the first programming languages revived the interest in extending the framework used to describe languages in mathematical logic to handle the syntax and the semantics of programming languages. The syntax of a programming language is formally described under the form of a grammar that contains the rules for the construction of programs. The semantics of a programming language is formally defined by a correspondence between the syntactic constructs and mathematical objects that describe the computation.

The compilation of programming languages, or the translation of a programming language to another programming language, can naturally be described based on the semantics correspondence: a compiler translates a program written in a source programming language to a program written in a target programming language conserving the semantics. The part of the compiler in charge with the recognition of the source language is commonly called the *front-end*. The term *back-end* is commonly used for the part of the compiler that generates the target language. The term *middle-end* is used for all the intermediate languages used in the compiler that are independent of the source and of the target languages. Modern compilers can compile several source languages to several target languages, because they internally use several layers of languages that are computationally equivalent to the source and to the target languages. The role of modern compilers is not only to translate the source to the target languages, but also to efficiently use the target languages, to exploit the architectural characteristics of the target computing machine. Thus, modern compilers implement some of the sophisticated optimizations initially introduced for supercomputing applications [AK02]: they provide performance models and transformations to improve fine-grain parallelism and exploit the memory hierarchy.

The GNU Compiler Collection (GCC) is the leading compiler suite for the free and open source software. The large set of target architectures and standard compliant front ends makes GCC the de facto compiler for portable developments and system design. However, until recently, GCC was not an option for high performance computing: it had no infrastructure neither for data access restructuring nor for automatic parallelization. Furthermore, until recently, the compilation research community had no robust, universal, free, and industry-supported platform where new ideas in compiler technology could be implemented and tested on large scale programs written in standard programming languages. For more than a decade, the field of advanced compilation has been plagued by redundant development efforts on short-lived projects, with little impact on production environments.

1.1 Motivations

Most of the industrial, commonly used programming languages are only defined based on informal natural language descriptions. Large parts of the semantics are intentionally left undefined, or left to be defined by the compiler. In turn, compilers, intermediate languages used in compilers, and compiler algorithms are informally described in the literature: a perpetrated tradition. This situation makes the definition of formal tools hard, especially in the case of compilers and verification tools. In the case of free software compilers, the source code itself provides a formal definition. In this respect, the free software compilers improved the situation over past descriptions of compilers. However, the source code of these compilers is too large, containing too many details and special cases. This thesis will provide an equivalent higher-level formal overview of some of the intermediate languages used in the free software compiler GCC. More specifically, this thesis will present for the first time a formal definition of the Static Single Assignment (SSA) language, and the intermediate languages that can be used for describing abstract properties of scalar variables extracted from the SSA representation, such as the envelopes of scalar evolutions.

Recent advances in compiler techniques show a preference on using higher level representations, in lieu of low level imperative languages, for performing code transformations and analyses: the use of single assignment languages [SS70] in compilers originates in the mid 1980s [AWZ88, RWZ88, CFR+89, CFR+91, Hav93, TP95b, TP95a, BP99, BP03], and as we will see in this thesis, these languages correspond to declarative languages, in which the notion of a store point, or memory cell, characteristic of imperative languages, is replaced by declarations and streams of values. Based on this higher

level description of programs, compilers are more apt to reorganize the code for efficiently translating programs on new architectures, and for proposing higher level static analysis techniques.

In opposition to dynamic analysis, testing or profiling, static analysis is performed without knowing the context in which the program will be executed. Thus, static analysis obtains more general solutions, but most of the interesting questions become uncomputable: every property that is not true or false for all the programs is undecidable, a result commonly known under the name of Rice's theorem [Rog87, Jon97]. In order to avoid uncomputable properties, a static analysis uses safe approximations of the behavior of programs, a trade between the cost of static computation and the precision of the provided answers. In some cases the abstract representations would not contain enough information for allowing the static analysis to output a definitive answer: the result is an uncertain answer, "don't know". Abstract interpretation [CC77] provides a general theoretical framework for the approximation of the semantics of programming languages. The abstract interpretation framework provides an automatic technique to build static analyses starting from the most precise semantics of the programming language and targeting some predefined abstract domain. The results of a static analysis are used in different applications, ranging from the detection of safety properties of programs, to validation of code transformations in loop nest optimizations and automatic parallelization [Wol96, Muc97, AK02].

1.2 Contributions of this Thesis

The main contributions of this work are to provide formal definitions and practical implementations of static analyses using the SSA form in industrial compilers. More precisely, the contributions of this thesis can be summarized in two parts: contributions to theoretical frameworks, and results from the practical implementations of these theoretical ideas in an industrial compiler (GCC).

The first theoretical contribution of the thesis consist in a formal definition of the SSA language under the form of a syntax and a denotational semantics. We provide also a minimal imperative language, that we called IMP, for which we provide the translation algorithm to the SSA language. The number of syntactic elements composing these languages has been intentionally reduced to a minimum, only preserving the vital parts of the languages that makes them computationally equivalent to any other full-fledged language. This reduction of the number of syntactic elements reduces the length of the proofs for the consistency theorem that states the soundness of

the translation algorithm from IMP to SSA.

The definition of the denotational semantics of the SSA language has a direct application in the extension of the applicability of the classical abstract interpretation framework to programs written in SSA form. This thesis provides several examples that illustrate the use of classical abstractions on the SSA representation. The thesis also presents a practical framework for prototyping static analyses of the SSA language: the framework is based on PROLOG and uses the full power of the unification algorithms provided by this language for implementing an interpreter for the SSA language, providing an operational semantics of the SSA language, and for the description of several algorithms that extract abstract views of the SSA language.

From a practical point of view, this thesis illustrates a part of the theoretical developments with industrial developments: the implementation of the static analysis algorithms described in this thesis in an industrial compiler, the GNU Compiler Collection. The implementation is freely available for reference in the main releases of GCC starting from versions 4.0. These static analyses are at the heart of the new loop nest optimization framework that we are currently developing in the GNU Compiler Collection. The thesis provides an evaluation of the importance of these static analyses in the context of these loop nest optimizations on different computer architectures.

1.3 Overview of the Thesis

The thesis is organized as follows: Chapter 2 recalls the previous definitions of the SSA, and then provides a formal definition of a syntax and a denotational semantics for the SSA language, finally the chapter concludes with the proof of a theorem stating the consistency of a translation algorithm from an imperative language to SSA. Chapter 3 defines a set of abstractions on top of the denotational semantics of the SSA, then more practically we will see a possible encoding of SSA programs in logic programming using PROLOG, and some techniques used to define static analyses on the SSA. Chapter 4 provides the imperative version of the main static analysis. This description of the same analysis algorithm is more practical for compiler writers that want to adapt these static analyses in their compiler framework: this description is closer to the implementation that I have integrated in GCC. The chapter concludes with several experiments that show the importance of this static analysis in the current optimization framework of GCC. Finally, Section 5 concludes and sketches future work.

Chapter 2

Denotational Semantics of the Static Single Assignment Form

The main contribution that I will describe in this chapter is related to the definition of a formal syntax and semantics of the SSA representation. I will survey different semantics that were given to the SSA language, and then, I will present the first denotational semantics of the SSA language. I will then describe a denotational semantics of the algorithm that converts an imperative language to the SSA form, and finally I will show that the translation is consistent with respect to a property that is conserved during the conversion. For this purpose, I will use the denotational recording semantics of an imperative language that I called IMP.

First, I will describe some formal methods that are available for defining a programming language.

2.1 Definition of a Programming Language

Programming languages can be distinguished from natural languages in that they are usually built using formal methods. In some cases, the use of an informal language for speaking about algorithms is useful for conveying an intuition, but is inappropriate for formal proofs. Programming languages can be synthesized from mathematical objects that describe their form, commonly called the syntax, and their meaning, called the semantics. We find this definition in [Rog87]:

> *Syntax is the study of formalized systems as pure formalisms, apart from intended meanings. Semantics (in logic) is the study of the relation between formalized systems and the mathematical objects (e.g. real numbers) about which the systems appear (or are*

intended) to speak. The distinction between syntax and semantics is intuitively useful but difficult to make fully precise.

As the main topic of this thesis is related to programming languages, I will use the term language exclusively for referring to a programming language. In the remaining of this section, I survey the existing tools commonly used to describe the syntax and the semantics of languages.

2.1.1 Syntax of a Language

In general, languages are encoded in streams of characters. A structure is given to this stream of letters, digits and other symbols by a lexical analysis that groups characters into tokens, that are then combined into more elaborated structures by the rules governing the syntax of the language. The syntax of a language is defined by describing the structure of the language in terms of a grammar. The grammar of a language contains the rules that can be used either to build new syntactic objects belonging to the language, or that can be used to verify the validity of a syntactic construct. The result of the syntax analysis is an Abstract Syntax Tree, or AST, that represents the tree structure of a part of the program that was analyzed. These notions were formalized in the 1950s and is nowadays called the BNF, the Backus Normal Form, or sometimes also called the Backus-Naur Form for also accounting the contributions of Peter Naur. A detailed overview of the techniques used to build lexical and syntax analyzers is described in [ASU86].

Once the syntax of a language is formally defined, the remaining task for building a translator or a compiler, is to give a correspondence between each component of the language and some other object: that can either be a mathematical object, or the constructs of another language. In the following, I will describe different means for defining such correspondences.

2.1.2 Semantics of a Language

Several techniques have been proposed to give a meaning to a language, as: state transitions, mathematical objects, language translations, i.e. compilers. Each of these techniques illustrates different aspects of a programming language, and thus the different semantics should not be seen as competing description techniques, but complementing each other.

In the following, I will give a brief overview of the different tools that are available to describe the semantics of a language.

Informal Semantics. The informal semantics gives the meaning of a language with informal definitions written in a natural language, potentially using examples with informal discussions and explanations of the operational mode.

Most of the introductory books for programming languages are based on informal semantics, as this provides intuitive descriptions that are simpler to be taught, simpler to understand, and simpler to learn than some abstract description, or mathematical model.

As a striking example, this is the only way programming languages are described in standards, leading to unspecified behaviors, ambiguities, and omissions. This is a painful situation that plagues the entire software industry: the construction of compilers that comply to the standard texts are difficult to build, as compiler engineers are faced to an interpretation problem of the standard. Programmers are faced to a double uncertainty problem, as they have to learn the subtleties of the wordings of the standard, and furthermore they have to learn the observational behavior of the compilers. On the other hand, from the point of view of the so called "language lawyers", members of a language standardization committee, this seems to be the only viable situation, as the complete specification of the language via the implementation in a non natural language of a "standard interpreter" or a "standard compiler" is completely inconceivable because of the tensions between the members of these committees that represent companies developing commercial compilers. The complete formal specification of a language can seem to be a tedious task, for example, when looking at the complete denotational semantics of the C language [Pap98], a tedious task that is not justified by any concrete need coming from the industrial world.

One of the causes of this situation comes from a rather profound fission between the software industry and the language research community. A typical example of this can be seen in one of the books that is nowadays considered as foundational to the compiler techniques from the number of citations that it has gathered during its first twenty years from its first publication: the so-called "Dragon Book" [ASU86]. The book clearly defines the syntax analysis, but remains completely vague on the semantics analysis, finishing by stating:

> *In this book we shall not attempt to define programming-language semantics formally at all, since none of these methods has gained universal acceptance. Rather we shall introduce terminology that lets us talk about the most common choices for the meaning of common programming constructs.*

This is a quite striking remark, but can be understood, for the main purpose

of the book is to provide an informal semantics of the fundamental analysis and transformation algorithms that constitute a compiler.

Another reason why the complete formal specification of industrial languages is difficult is due to their complexity: John Backus, one of the first pioneers in language design, who designed among others the FORTRAN language, has stated in the Turing Award Lecture of 1977 [Bac78] this increasing complexity trend in language design:

> *Conventional programming languages are growing ever more enormous, but not stronger. Inherent defects at the most basic level cause them to be both fat and weak: their primitive word-at-a-time style of programming inherited from their common ancestor – the von Neumann computer, their close coupling of semantics to state transitions, their division of programming into a world of expressions and a world of statements, their inability to effectively use powerful combining forms for building new programs from existing ones, and their lack of useful mathematical properties for reasoning about programs.*

A second example of the fission between software industry and language research is that most of the papers related to intermediate representations of compilers are written by and for compiler engineers. They completely overlook the fact that intermediate representations are languages, and most of the time provide informal semantics for their algorithms. These informal descriptions certainly are appropriate to convey some information, avoiding implementation details, but at the same time, these descriptions are not enough precise, and cannot be used in standard formal frameworks.

The main topic of this chapter illustrates these ideas, by proposing, for the first time, a formal definition for the Static Single Assignment SSA form, an intermediate representation used for more than a decade in most of the industrial compilers. The justification of the need of such a formalism is given with respect to the insights that the denotational semantics of the SSA language brings to the understanding of the structure of the SSA language.

Denotational Semantics. The denotational semantics has been proposed by Christopher Strachey [SM76], and is based on Dana Scott's mathematical developments of domain theory [Sco82]. Each language construct is given a meaning using a denotation, that is a mathematical object modeling the language constructs. The denotational semantics is similar to the syntactical structural definition of the BNF, in that its aim is to describe the meaning of a syntactic construct in function of the meaning of each of its syntactic

components. The denotational semantics describes concepts structured as parts of programs rather than the atomic operations that a machine could have to execute. An often used image for the standard semantics is that of "big steps": meaning is given to large chunks of programs that are structured as in an AST, potentially containing several levels of components. As these descriptions are built hierarchically, the denotational semantics descriptions are modular and concise.

Continuation-Passing Style. In order to encode the notion of control flow in an imperative language, it is possible to use a "continuation" that is in general a function that encodes the state of the system during its execution. The continuation function will hold the "returning" value once the program has been executed. This technique is often used in denotational semantics for dealing with complicated control flow constructs such as the "goto statements" in imperative programs.

Operational Semantics. The operational semantics gives a very intuitive meaning, as it is a mostly mechanical operational mode of a program: describing the effects of the execution of a command on the states of a machine. Usually the semantics of a language is specified by a set of transition rules that transform the state of the machine. As the specification of an operational semantics is based on the specificities of a target machine, abstract machines have been proposed to ensure the portability of the operational semantics on real machines, as for example in the case of the Java technology. A particular kind of abstract machine is the Abstract State Machine ASM that defines, based on the AST of the program, state transitions as evolving algebras [Gur95]: a basis algebra is extended by the execution of a syntactic construct of the language. The collection of all the rules that extend the basis algebra is the semantics of the language. This provides an alternate view of the operational semantics of the language.

Recording Semantics. The recording semantics, gathers some information available while executing the program, under the form of traces of execution that record all the intermediate states of the machine during the evaluation of the program. For example, we will use such a recording semantics for an imperative language, recording for each identifier and at each store point in the program, a value. The store points in an imperative language can be static, as in a sequence of assignments, and dynamic, as in the successive evaluations of a same assignment in a loop.

Indirect Semantics. The indirect semantics gives the meaning of a translation specifying the relation between the semantics of the source and the target language representations. The usual example that illustrates the indirect semantics is that of a compiler: starting from an intermediate representation of a program, a compiler translates that program to another language, with different syntactic constructs and different meaning.

Axiomatic Semantics. The axiomatic semantics has been proposed by Hoare [Hoa69], and formulates the meaning of a language under the form of transformation effects of logic propositions. The meaning of a program is not explicitly given by the axiomatic semantics, but is the result of the combination of several transformation rules on the given preconditions. The information obtained in the postconditions for some language construct and some preconditions, can be incomplete, if only a part of the transformation rules have been applied.

Several of these semantics can be used when formalizing the meaning of a language for describing different aspects of the same constructs. The operational semantics would provide an interpretative view suited to the construction of an interpreter of the language. The denotational semantics being structural is suited to the construction of a compiler, or of a structural analyzer. The following sections will examine different semantics of a representation widely used in compiler technology: the Static Single Assignment (SSA) form.

2.2 The SSA Intermediate Representation

The SSA representation has been proposed in the late 1980s [CFR+91] for speeding up data flow analysis algorithms. Twenty years later, this intermediate representation has been adopted by a wide range of compilers from the industrial optimizing compilers (GCC [GCC05], Intel CC [ICC]) to academic compilers (LLVM [LA04], HiPE [LPS05]). At the origin, the SSA form was primarily used in compilers for imperative languages, but after the proposition of an equivalence between the SSA representation and the continuation-passing style [Kel95], the SSA form has also been successfully used as a target representation for functional languages [LPS05].

In the following, we will survey the semantics of the SSA language that have been proposed in [CFR+91, Kel95, Gle04], then we will see a completely new semantics for the SSA form that we proposed in [PCJS06a], a conversion algorithm from an imperative language to the SSA form, and a proof of

correctness for this algorithm. The next section will introduce some classic terminology that is needed in the following presentations.

2.2.1 Basic Notions of Data Flow Analysis

The main ideas used in imperative program analysis can be found in classical textbooks [ASU86, Muc97, NNH99], and we shall review here only the basic notions that will be extensively used in the remaining of this thesis.

The Control Flow Graph. The control structure of imperative programs is usually represented using a directed graph, in which edges stand for possible transitions between nodes, called basic blocks, that contain the commands to be executed. Inside basic blocks there is no control structure other than the implicit sequence, that is, if the first command of a basic block is executed, then all the other commands of the basic block are executed sequentially one after the other, except in the case of an error. This graph is called the Control Flow Graph or CFG, and is a common representation in modern compilers, where it sometimes completely replaces high level control constructs, such as the if, switch, loop constructs and exceptions. Figure 2.1 illustrates the construction of the CFG with an example, in which an imperative program is first lowered to a form with jumps and labels, i.e. the syntactic control structure is removed, and then the control flow graph is built on top of this representation, providing a higher level structure that can be faster traversed than the text of the unstructured program.

On top of this representation, it is usual to build other higher level representations, such as the natural loops [ASU86], and the SSA representation that will be described in the next sections. Another application that we will survey next is the data flow analysis, in which data flow problems are commonly set as equations at the boundaries of basic blocks.

Data Flow Analysis. The information extracted by a data flow analysis represents how the program manipulates its data. There are several kinds of data flow analysis that share the same iterative analysis framework. A data flow problem is stated by a set of data flow equations classically represented with bit vectors. The solution to this system of equations is computed iteratively, leading to a chain of solutions of increasing precision.

Many analysis problems can be simply formulated in the data flow framework, and the SSA representation corresponds to an improvement of this infrastructure: the SSA intermediate representation was principally proposed in compilers for imperative languages as a data structure that reduced the

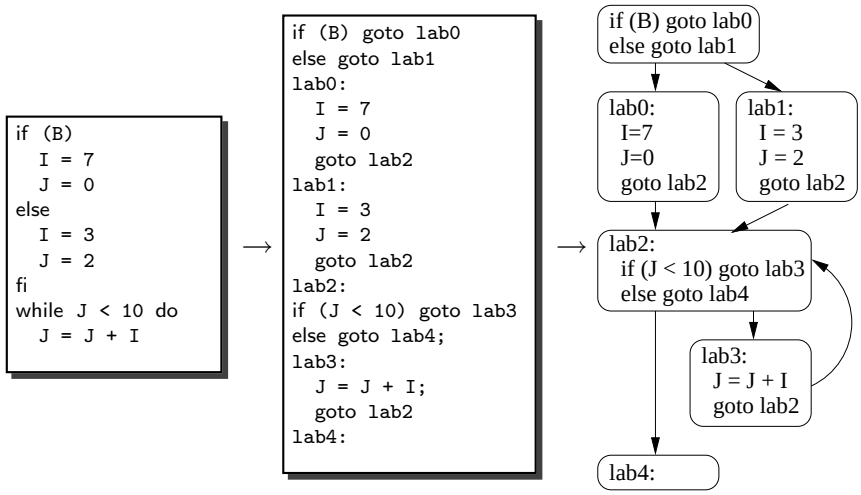


Figure 2.1: Translation of an imperative program to CFG.

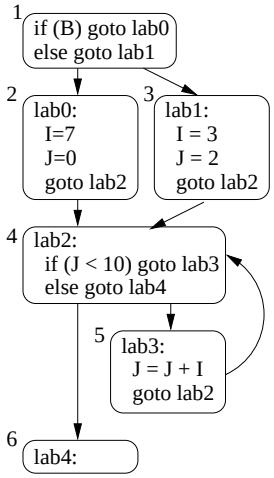
complexity of data flow analysis algorithms. The complexity reduction was mainly due to a radical change in the data structures used for encoding the data flow information: classical data flow algorithms propagate bit vectors through the structure of the imperative program. The complexity of accessing some information in bit vectors is $\mathcal{O}(\log(n))$ for some bit vector containing n entries. Where a classical data flow analysis would have taken $\mathcal{O}(n \log(n))$, the equivalent algorithm based on the SSA form performs the same analysis in $\mathcal{O}(n)$ [CFR+91], as the algorithm based on the SSA form performs the analysis on a single variable at a time, avoiding the use of bit vectors for encoding the data flow information.

Reaching Definitions Analysis. An example of data flow analysis is the reaching definitions analysis. The notion of reaching definitions analysis is proper to the imperative languages, and should more appropriately be called "reaching assignment analysis" as its main purpose is to identify the last assignment command that modified the value of a store. The identification is generally based only on a syntax numbering of assignment commands. Figure 2.2 illustrates on the CFG of the previous example the construction of the data flow equations needed to solve the reaching definitions problem. The notation $RD_{exit}(x) = RD_{entry}(x)[l/v]$ updates to l the value associated to the variable v in the function RD_{entry} for basic block x. The $\cup$ operator merges the sets of two branches, and more formally the RD_{entry} and RD_{exit}

relations are defined as:

$$\begin{aligned} RD_{entry}(x) &= \begin{cases} \{(i, \perp) \mid i \in Ide\}, & \text{if } x = init, \\ \bigcup\{RD_{exit}(x_1) \mid x \in flow(x_1)\}, & \text{otherwise.} \end{cases} \\ RD_{exit}(x) &= RD_{entry}(x)[x/v]_{v \in def_x} \end{aligned}$$

where Ide are a set of identifiers used in the program, $init$ represents the entry basic block, $\perp$ is the empty set, and the relation $flow$ is defined by the structure of the CFG. We also use the notation $[\,]_{v \in def_x}$ for iterating over all the variables v defined, or more correctly assigned to, in basic block x.



$RD_{entry}\ (1) = \{(\text{I}, \perp), (\text{J}, \perp)\ \}$
$RD_{entry}\ (2) = RD_{entry}\ (3) = RD_{exit}\ (1)$
$RD_{entry}\ (4) = RD_{exit}\ (2) \cup RD_{exit}\ (3) \cup RD_{exit}\ (5)$
$RD_{entry}\ (5) = RD_{entry}\ (6) = RD_{exit}\ (4)$
$RD_{exit}\ (1) = RD_{exit}\ (1)$
$RD_{exit}\ (2) = RD_{entry}\ (2)\ [2/\text{I}]\ [2/\text{J}]$
$RD_{exit}\ (3) = RD_{entry}\ (3)\ [3/\text{I}]\ [3/\text{J}]$
$RD_{exit}\ (4) = RD_{exit}\ (4)$
$RD_{exit}\ (5) = RD_{entry}\ (5)\ [5/\text{J}]$
$RD_{exit}\ (6) = RD_{entry}\ (6)$

Figure 2.2: Reaching definitions analysis as a data flow problem.

As the above equations are recursive, it is possible to set the problem as:

$$(RD_{entry}(1), \ldots, RD_{exit}(6)) = F(RD_{entry}(1), \ldots, RD_{exit}(6))$$

and then, the reaching definitions data flow problem is solved by an iterative computation as the least fixed point for function F.

A more precise definition of the reaching definition could include, in addition to the syntactic coordinate, an iteration space numbering, in which case, there are two distinct components for the identification of an assignment: a purely static syntactic coordinate that points in the abstract syntax tree, and a dynamic coordinate that points to the iteration numbering of loops [Ami04, ACF06]. One of the main innovations of the SSA technology is that the static coordinate completely disappears, and sole remains the dynamic part that cannot be represented statically. We shall very clearly see this

improvement in Section 2.3.4, that states the theorem of consistency of the translation of an imperative language to SSA form.

Now that we have surveyed some notions about the classical imperative program analysis, I will present the algorithms proposed in [CFR+91] that first shown a fast translation into and out of the SSA, making this representation affordable in industrial compiler frameworks.

2.2.2 An Informal Semantics for SSA

Several papers have described the construction [CFR+91, BP03, BCHS98], analyses and optimizations [AWZ88, RWZ88, WZ91] based on the SSA form. The following presentation will mostly be based on [CFR+91].

A Rough Definition. Although many compiler textbooks describe the SSA representation [Wol96, Muc97], no formal definition is given, so we just provide the same informal, very crude definition of the SSA form: a program is in SSA form if every variable used in the program appears a single time in the left hand side of an assignment. In order to intuitively explain this definition, several examples of the translation to SSA form are provided, eventually ending on a CFG based algorithm that constructs the SSA form. I will exactly follow this informal way of presentation of the SSA form: Figure 2.3 illustrates a basic case of translation to SSA form for the sequence. The main

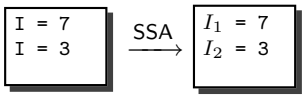


Figure 2.3: Translation of a sequence to SSA form.

problems in this renaming process occur at control flow merge points, where several definitions have to be merged to a unique name that can be used in the rest of the program. Figure 2.4 illustrates this dilemma.

ϕ Nodes. In order to merge the information coming from different control flow branches, the SSA representation introduces new assignment statements, called ϕ (phi) nodes. The efficient placement of these extra assignments is extensively discussed in the literature [CFR+91, BP03], and these algorithms are based on the notion of dominance frontiers.

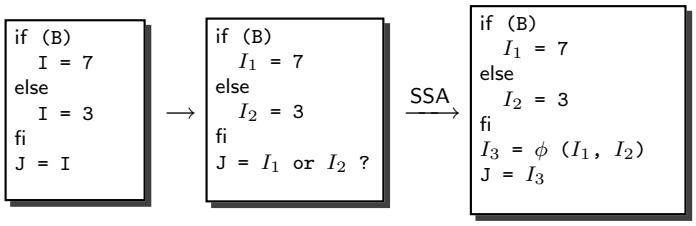


Figure 2.4: Translation problem for condition statements.

Dominance Frontiers. The dominance frontiers is a mapping from basic blocks to a set of basic blocks that is computed on top of the dominator tree. The dominator tree is a data structure that provides an efficient access to the dominance relation [Tar74] defined by $X \gg Y$ iff "X appears on every path from Entry to Y" in a CFG with an initial node "Entry". The dominance frontiers is then defined as:

$$DF(X) = \{Y \mid \exists P \in Pred(Y), X \gg P \wedge \neg(X \gg Y)\}$$

for some basic block X, with $Pred(Y)$ the set of predecessors of basic block Y.

Placing ϕ Nodes. The ϕ nodes have to be placed at control flow joins, that are blocks in the CFG where two or more distinct paths join. The set of join points for two distinct blocks in a subset S of a CFG is given by the function $J(S)$. The iterated join $J^+(S)$, computed iteratively as the limit of the increasing sequence,

$$\begin{aligned} J_1 &= J(S) \\ J_{i+1} &= J(S \cup J_i) \end{aligned}$$

is shown to be equal [CFR+91] to the iterated dominance frontiers $DF^+(S)$ defined as the limit of the increasing sequence:

$$\begin{aligned} DF(S) &= \bigcup_{X \in S} DF(X) \\ DF_1 &= DF(S) \\ DF_{i+1} &= DF(S \cup DF_i) \end{aligned}$$

For some variable V, let S be the set of basic blocks that contain definitions of V. The phi node insertion points are given by $DF^+(S)$. Once the phi nodes have been inserted for each variable defined in the program, the variables are renamed and that finishes the construction of the SSA form.

Discussion. As can be remarked in this presentation, as well as in almost all the papers defining the construction of the SSA form, the semantics of the SSA language appears only as a byproduct of the algorithms that are building this structure. One of the characteristics of these presentations is that they are basing their proofs on graph theory, as the CFG based language that they consider has lost its syntactic structure. A structural based construction of the SSA form has been described in [BM94], but the presentation remains very practical. Furthermore, instead of defining a language, as later defined by [Kel95], these early papers informally present the conversion algorithms, focusing on the practical aspects, and on the ease of implementation in a compiler.

It is possible to consider the early papers on SSA form as means to make the SSA form widely accepted in the imperative compilers community, based on the practical strengths that the SSA representation has shown in the early experiments. This also explains the important impact of [CFR+91] that was the first presentation of a fast practical algorithm to construct the SSA form, and to translate the SSA form back to the original imperative language.

The translation algorithm is simple enough to appear intuitively correct, and the construction of the SSA form is not proved to be consistent with respect to some starting representation. Although a more formal presentation would have allowed a proof of the translation correctness, the early forms of presenting the SSA language are not proper to this kind of proofs.

In the following, we shall see a formal indirect semantics for the SSA, given by Kelsey in [Kel95] under the form of two translation algorithms between the continuation-passing style CPS and the SSA form. Although, the goal of Kelsey is the compilation of functional programs, his framework could have been appropriate to formally prove the consistency of the translation to SSA form. So the main contribution of [Kel95] stands in improving our knowledge of the SSA language by providing the compilers SSA $\rightarrow$ CPS and CPS $\rightarrow$ SSA.

2.2.3 SSA is Functional Programming: SSA = CPS

The continuation-passing style is a technique used in functional programming languages for specifying the control flow in a computation. A continuation

is a function that contains the code to be executed on the result of a computation, and thus in continuation-passing style, the computation is performed only after a trace of execution has been collected. A similar technique is known under the name of tail call: the CPS corresponds to a tail call in which the function to be executed, when ending the execution of the current function, is passed as a parameter.

Using these techniques, it is possible to simulate the execution of arbitrary control flow, and in particular, Kelsey has explicitly shown in [Kel95] the translation algorithms that we will quickly survey in the following.

A Flowchart Syntax for SSA Representation. One of the first presentation of the SSA form as an imperative language satisfying the unique assignment for the variables, is given by a flowchart language containing classical imperative constructs, with the control flow explicitly expressed by goto and if statements. In this presentation, we will see a reduced syntax and a reduced algorithm that is enough for giving the main idea behind the pioneering presentation of Kelsey.

The syntax of the SSA language can be defined as a collection of labels L, followed by a sequence of phi node declarations I, that are in the head of a basic block that contains a sequence of instructions defining the body of the block: B. The language also contains a very restricted set of constructs for expressions E.

$$\begin{array}{rcl} L & ::= & l : I^* B \\ I & ::= & x \leftarrow \phi(E^*) \\ B & ::= & x \leftarrow E; B \mid \mathsf{goto}\ l_i \mid \mathsf{if}(E)\ B\ \mathsf{else}\ B \\ E & ::= & x \mid E + E \end{array}$$

The semantics of this language is not specified in [Kel95], but is left under the informal sentence:

> *The semantics is the 'obvious' one.*

However Kelsey provides a translation algorithm from a subset of Scheme, that gives this language an indirect semantics.

Kelsey do not intended to prove any consistency property of the conversion algorithms, but the originality of the work of Kelsey stands, in fact, in the enunciation of the translation algorithms.

A Syntax for CPS. As in the previous paragraph, we shall see a reduced set of the syntax originally given by Kelsey: small enough for making this presentation concise, but powerful enough for expressing the main idea behind the conversion algorithms. The syntax of CPS is mostly the syntax of a functional language, with

$$\begin{array}{rcl} P & ::= & (\lambda_{jump}(x^*)M) \\ M & ::= & (\mathsf{if}\ E\ M\ M) \mid (let\ ((x\ E))\ M) \mid (J\ E^*) \\ E & ::= & x \mid E + E \end{array}$$

In this case again, Kelsey omits the definition of a semantic for this language that is left under the informal sentence:

> *The semantics is the 'obvious' call-by-value semantics.*

This indeed is the semantic of a well known construct also known under the name of "call-by-worth", and can be found in denotational semantics textbooks [SM76, Sto77, Gor79]: informally, the semantics corresponds to the evaluation of the parameters before the evaluation of the call.

The following two algorithms provide the relation between the syntactic objects of the SSA and CPS languages.

From CPS to SSA. The function $\mathcal{G}_{jump}[\![]\!]$ translates a jump abstraction from CPS to a labeled basic block in SSA. The function $\mathcal{G}[\![]\!]$ is used to convert the code contained in the jump abstraction from CPS syntactic constructs (M) to SSA syntactic constructs (B).

$$\begin{array}{rcl} \mathcal{G}_{jump}[\![]\!] : J \times P & \rightarrow & L \\ \mathcal{G}_{jump}[\![J, (\lambda_{jump}(x\ldots)M)]\!] & = & J : x \leftarrow \phi(E_{0,0}, E_{0,1}, \ldots); \ldots \mathcal{G}[\![M]\!] \\ \mathcal{G}[\![]\!] : M & \rightarrow & B \\ \mathcal{G}[\![(let\ ((x\ E))\ M)]\!] & = & x \leftarrow E; \mathcal{G}[\![M]\!] \\ \mathcal{G}[\![(J\ E_{0,i}\ E_{1,i}\ \ldots)]\!] & = & \mathsf{goto}\ J_i \\ \mathcal{G}[\![(\mathsf{if}\ E\ M_1\ M_2)]\!] & = & \mathsf{if}(E)\ \mathcal{G}[\![M_1]\!]\ \mathsf{else}\ \mathcal{G}[\![M_2]\!] \end{array}$$

From SSA to CPS. The function $\mathcal{H}_{jump}[\![]\!]$ translates a labeled basic block from SSA to a jump abstraction in CPS, using the $\mathcal{H}[\![]\!]$ function that converts

an SSA block of instructions to CPS.

$$
\begin{aligned}
\mathcal{H}_{jump}[\![]\!] : L &\rightarrow P \\
\mathcal{H}_{jump}[\![J : x \leftarrow \phi(E_{0,0}, E_{0,1}, \ldots); \ldots B]\!] &= (\lambda_{jump}(x \ldots)\mathcal{H}[\![B]\!]) \\
\mathcal{H}[\![]\!] : B &\rightarrow M \\
\mathcal{H}[\![x \leftarrow E; B]\!] &= (let((xE))\mathcal{H}[\![B]\!]) \\
\mathcal{H}[\![\mathsf{goto}\ J_i]\!] &= (J\ E_{0,i}\ E_{1,i}\ \ldots) \\
\mathcal{H}[\![\mathsf{if}(E)\ B_1\ \mathsf{else}\ B_2]\!] &= (\mathsf{if}\ E\ \mathcal{H}[\![B_1]\!]\ \mathcal{H}[\![B_2]\!])
\end{aligned}
$$

Kelsey gives an informal definition for the labels used in the translation algorithms:

> *The $E_{i,j}$ on the right-hand side of the definition of $\mathcal{H}[\![\mathsf{goto}\ J_i]\!]$ are those from the left-hand side of the definition of $\mathcal{H}_{jump}[\![]\!]$.*

Discussion. This presentation of the equivalence between SSA and CPS languages has been extracted from [Kel95] in a condensed form, for illustrating the originality of this work that was the first to detach from the classical presentations of the SSA form.

The original presentation offers a wider range of syntactic objects, and is intended to become a starting implementation of a real compiler. Indeed, the main motivation behind the work of Kelsey in [Kel95] is the compilation of functional languages through the optimizing SSA compiler infrastructure. The other way around, translating SSA to CPS, provides some inspiration for interprocedural optimizations on SSA form, mainly based on the advanced techniques existing in compiling functional languages.

We recognize the importance of the work of Kelsey, as it states the "folklore" understanding of the connections between SSA and functional programming [Kel95]. First, the semantics of the SSA and CPS languages are not provided, but this is not critical since their semantics are close to those of the flowchart and of the functional languages available in textbooks. More difficult questions are left unsolved, as the consistency of the composition of the translation algorithms: one could expect to obtain an idempotent operation by the application in sequence of the conversion from CPS to SSA form and then back to CPS form. The translation algorithms also need notational improvements, ans miss critical steps (ellipses in the original presentation).

2.2.4 Discussion on Related Work

Since the motivation for the introduction of the SSA representation is mostly one built out of experience stemming from the implementation of compilers' middle ends, there is scant work looking at its formal definition and properties. Yet, as we have seen, some previous work offers a couple of different semantics for the SSA form:

- The early papers [CFR+89, CFR+91], which introduce the notation for SSA, mostly present informal semantics and proofs for some optimization algorithms based on the SSA representation.

- Kelsey [Kel95] studies the relationship between the SSA form and the functional programming paradigm, providing a somewhat more formal view of the semantic link between these two notions. He defines an indirect semantics that translates programs in SSA form to continuation-passing style and back to SSA, providing a way to compile functional languages to the SSA, and making it possible to use the SSA optimizing technology on functional languages. In some sense, our work can be viewed as opening a new venue for this approach by formally showing that the imperative programming paradigm can be mapped to the SSA form.

- A similar semantics based on continuations is given by Glesner [Gle04]: she gives an abstract state machine semantics for the SSA, and uses an automatic proof checker to validate optimization transformations on SSA. Yet, there is no formal proof provided to ensure the correctness of this mapping between ASM and SSA. In the following section, we will see a different, denotational, semantics for SSA and use it to prove the correctness of the SSA conversion process for imperative programs.

All the existing definitions of the SSA form in the literature are influenced by the early papers [CFR+91] and consider the SSA as a data structure on top of some intermediate representation, e.g., control-flow graphs augmented with a stream of commands belonging to some imperative language, in other words, a decoration on top of some existing compiler infrastructure. In contrast, the following presentation is the first to give a complete definition of the SSA form, promoting the SSA to the rank of a full-fledged language. An important aspect of the SSA is exposed this way: the SSA is a declarative language, which, as such and contrarily to what its name might imply, has nothing to do with the concept of assignments, a notion only pertinent in imperative languages. This declarative nature can explains why the SSA

language is particularly well-suited to specify and implement program optimizations.

2.3 A Denotational Semantics for SSA

Most modern and widely distributed compilers for imperative and even some functional languages use the SSA form as an intermediate code representation formalism. This Static Single Assignment format [Muc97] is based on a clear separation of control and data information in programs. While the data model is data flow-based, in such a way that no variable is assigned more than once, the control model traditionally is graph-based, and encodes basic blocks linked within a control-flow graph. When more than one path reaches a given block, values may need to be merged; to preserve the functional characteristics of the data flow model, this is achieved via so-called ϕ-nodes, which assign to a new identifier two possible values, depending on the incoming flow path.

If this formalism is successfully used in both academic (e.g. GCC [GCC05], LLVM [LA04]) and commercial (Intel CC [ICC]) compilers, we believe its theoretical foundations are somewhat lacking: Section 2.2 surveyed some of the earlier attempts to formally describe such a framework. One of the main goals, in the following, is thus to provide what we believe to be a firmer foundation for this ubiquitous intermediate representation format, addressing both the SSA language in itself and the conversion process used to translate imperative source code to intermediate SSA constructs.

But, our approach is also practical in that we want to address one shortcoming we see in most of the current literature on the SSA form. The original motivation for the introduction of ϕ-nodes was the conditional statements found in imperative programming languages, for which two paths need to be merged when leaving the branches of an alternative. Thus, most of the methods of ϕ-node placement present in the literature omit the related but somewhat different ϕ-nodes that should also logically occur after loops. This is, in practice, not an issue, since most compilers' middle ends keep control-flow information on the side (e.g., control-flow graphs or continuations) to deal with loop exit conditions.

It is only quite recently, in the GCC community [Dvo, Zadb], that the need to introduce such additional ϕ-nodes became apparent, in particular when designing algorithms working directly on loop structures. The initial motivation for the use of these extra nodes was thus mostly practical [Zada] since they simplified the implementation of some code transformation techniques that required insertion of new edges in the SSA graph structures. This hum-

ble beginning may actually even explain why they were overlooked in recent surveys [BP03], as their role was yet not well understood at the time.

As we shall see in this presentation, these "loop-closing ϕ" expressions are in fact crucial to the expressiveness of SSA, providing the key construct that boosts the computational power of the "pure" SSA language, namely a functional data flow language without additional ad-hoc control-flow information, from primitive recursion to full-fledged partial recursive functions theory. Moreover, the structural nature of the denotational framework we use here, in lieu of the traditional graph-based algorithms in which the distinction between conditional and loop-originating edges is lost, make this requirement even more compelling.

2.3.1 IMP, an Imperative Programming Language

Since we are only interested by the basic principles underpinning the SSA conversion process, we will use a very simple, yet Turing-complete, imperative language, IMP, based on assignments, sequences and while loops. As is well-known, conditional statements can always be encoded by a sequence of one or two loops, and thus need not be part of our core syntax.

Syntax. IMP is defined by the following syntax:

$$\begin{array}{rcl} N & \in & Cst \\ I & \in & Ide \\ E & \in & Expr ::= N \mid I \mid E_1 \oplus E_2 \\ S & \in & Stmt ::= I = E \mid S_1; S_2 \mid \mathsf{while}_\ell\ E\ \mathsf{do}\ S \end{array}$$

with the usual predefined constants, identifiers and operators $\oplus$. Note that each while loop is decorated with a number label ℓ that uniquely identifies it. This labeling operation is assumed to have been performed at parsing time in a sequential manner[1]; all while loops are thus numbered, say from 1 to m, in a sequential fashion, for identification purposes. We only require that this numbering preserves the sequential textual order of the program. In the rest of the chapter, without loss of generality, we assume that the programs under study have a fixed number m of while loops.

[1]To conform to our denotational framework, note that this global decoration of the abstract syntax tree can also be specified as a denotational process.

Since the SSA semantics encodes recursive definition of expressions in a functional manner (see Section 2.3.2)), we found it easier to define the semantics for IMP as a recording semantics. It gathers, for each identifier and program point, its value during evaluation. To keep track of such evaluation points, we use both syntactic and iteration space information. Each statement in the program tree is identified by a Dewey-like number, $h \in N^*$; these numbers can be extended as $h.x$, which adds a new dimension to h and sets its value to x. For instance, the top-level statement is 1, while the second statement in a sequence of number h is $h.2$. The statement that directly syntactically follows h is $h+$, and is defined as follows:

$$\begin{aligned} 1+ &= 2 \\ (h.1)+ &= h.2 \\ (h.2)+ &= h+ \end{aligned}$$

Walking over an abstract syntax tree can be described with operations on a stack: for example, the successor of $h.2$, i.e. $(h.2)+$, the first operation pops the last digit, and then takes the successor of h, i.e. $h+$.

To deal with the distinct iterations of program loops, we use iteration space vectors k: their components represent the values k_ℓ of the m while loop indices at a given execution point (informally, k records all loop counter values). During evaluation, these vectors are modified, and we note $k[a/\ell]$ the vector obtained from k by replacing the value at index ℓ with a.

To sum up, a program evaluation point p is a pair $(h, k) \in P = N^* \times N^m$ that represents a particular "run-time position" in a program by combining both a syntactic information, h, and a dynamic one, k, for proper localization.

The only requirement on points is that they be lexicographically ordered, with the infix relation $<\in N^* \times N^* \to \mathit{Bool}$ such that $(h_1, k_1) < (h, k) = (k_1 < k \vee (k_1 = k \wedge h_1 < h))$. For any ordered set S, we note $\max_{<x} S$ the maximum element of S that is less than x (or $\perp$ if no such element exists).

Different other methods for identifying points in the program execution include traces and timestamps [Ven02]. In the following presentation we will exclusively use the above presented technique as all these previous techniques fail to separate program execution locations into syntactic and dynamic locations.

Semantics. As usual, the denotational semantics of IMP operates upon functions f on lattices or cpos [Sto77]; all the domains we use thus have a $\perp$ minimum element. Following a general convention, we note $f[y/x] = \lambda a.(y \text{ if } a = x, f(a) \text{ otherwise})$ and $f[z/y/x] = \lambda a.\lambda b.(z \text{ if } a = x \wedge b =$

$y, f(a,b)$ otherwise) the functions that extend f at a given point x. The domain of f, i.e., the set of values on which it is defined, is given as $Dom\ f = \{x \mid f(x) \neq \bot\}$.

The semantics of expressions uses states $t \in T = Ide \to P \to \mathcal{V}$; a state yields for any identifier and evaluation point its numeric value in $\mathcal{V}$, a here unspecified numerical domain for the values. The use of points gives our semantics its recording status; in some sense, our semantics specifies traces of computation. The semantics $\mathcal{I}[\![]\!] \in Expr \to P \to T \to \mathcal{V}$ expresses that an IMP expression, given a point and a state, denotes a value in $\mathcal{V}$ (we use $in_{\mathcal{V}}$ as the injection function of syntactic constants in $\mathcal{V}$):

$$\begin{aligned} \mathcal{I}[\![N]\!]pt &= in_{\mathcal{V}}(N) \\ \mathcal{I}[\![I]\!]pt &= R_{<p}(tI) \\ \mathcal{I}[\![E_1 \oplus E_2]\!]pt &= \mathcal{I}[\![E_1]\!]pt \oplus \mathcal{I}[\![E_2]\!]pt \end{aligned}$$

where the only unusual aspect of this definition is the use of the value of the reaching definition on a given function f, that we note $R_{<x}f = f(\max_{<x} Dom\ f)$. To obtain the current value of a given identifier, one needs to find in the state the last program point prior to the current p at which I has been updated; since we use a recording semantics, we need to "search" the states for this last definition.

The semantics of statements $\mathcal{I}[\![]\!] \in Stmt \to P \to T \to T$ yields the state obtained after executing the given statement at the given program point[2], given an incoming state:

$$\begin{aligned} \mathcal{I}[\![I = E]\!]pt &= t[\mathcal{I}[\![E]\!]pt/p/I] \\ \mathcal{I}[\![S_1; S_2]\!]p &= \mathcal{I}[\![S_2]\!](h.2, k) \circ \mathcal{I}[\![S_1]\!](h.1, k) \end{aligned}$$

These definitions are rather straightforward extensions of a traditional standard semantics to a recording case. For an assignment, we add a new binding of Identifier I at Point p to the value of E. A sequence simply composes the transformers associated to S_1 and S_2 at their respective points $h.1$ and $h.2$. And, as usual, we specify the semantics of a while loop as the least fixed point $\mathrm{fix}(W)$ of the W functional defined as:

[2]Remember that $p = (h, k)$.

$$\mathcal{I}[\![\mathsf{while}_\ell\ E\ \mathsf{do}\ S]\!](h,k) = \mathrm{fix}(W)(h,k[0/\ell])$$
$$W = \lambda w.\lambda(h,k).\lambda t.\begin{cases} w(h,k_{\ell+})(\mathcal{I}[\![S]\!](h.1,k)t), & \text{if } \mathcal{I}[\![E]\!](h.1,k)t, \\ t, & \text{otherwise.} \end{cases}$$

where, as a shorthand, $k_{\ell+}$ is the same as k, except that the value at index ℓ is incremented by one (we use latter $k_{\ell-}$, with a decrement by one).

If the value of the guarding expression E is true, we iterate the while loop at an updated point, which uses the same syntactic label as before, i.e. the syntactic beginning of the loop body $h.1$, but an iteration space vector where the value at index ℓ has been incremented, i.e. $k_{\ell+}$, since an additional loop iteration is taking place. If the loop test is false, we simply consider the loop as the identity function.

Example. To illustrate our results, we will use a single example; we provide in Figure 2.5 this simple program written in (a concrete syntax of) IMP, together with its semantics, i.e., its outgoing state when evaluated from an empty incoming state.

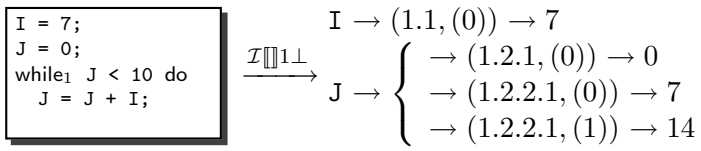


Figure 2.5: Syntax and semantics for an IMP program.

In this example, if we assume that the whole program is at syntactic location 1, then the first statement is labeled 1.1 while the rest of the sequence (after the first semi-column) is at 1.2; the whole labeling then proceeds recursively from there. Since there is only one loop, m is 1, and the iteration space vectors have only one component, initialized to (0). Thus, for instance, after two loop iterations, the value of J is 14, and this will cause the loop to terminate. The recording nature of the semantics is exemplified here by the fact we keep track of all values assigned to each variable throughout the whole computation.

2.3.2 SSA

In the standard SSA terminology [CFR+91, Muc97], an SSA graph is a graph of use-to-definition chains in the SSA form. Each assignment targets a unique variable, and ϕ nodes occur at merge points of the control flow to restore the flow of values from the renamed variables.

Here, we replace this traditional graph-based approach with a programming language-based paradigm; in the SSA form defined below, the ϕ declarations are capturing the characteristics of the control flow, and the usual control-flow primitives become consequently redundant. The use of this self-contained format is one of the new ideas we provide in this presentation, which paves the way to a more formal approach to SSA definition, its conversion processes and their correctness.

Syntax. A program in SSA form is a set of declarations of SSA identifiers $I_h \in Ide_{\mathsf{SSA}}$ to SSA expressions $E \in SSA$. These expressions are defined as follows:

$$E \in SSA ::= N \mid I_h \mid E_1 \oplus E_2 \mid \mathsf{loop}_\ell\phi(E_1, E_2) \mid \mathsf{close}_\ell\phi(E_1, E_2)$$

which extend the basic definitions of $Expr$ with two types of ϕ expressions. Note that identifiers I_h in an SSA expression are elements of Ide_{SSA} labeled with a Dewey-like number. Since every assignment in IMP is located at a unique h, this trick ensures that no identifiers in an imperative program will ever appear twice once converted to SSA form.

Since we stated that imperative control flow primitives are not part of the SSA representation, we intendedly annotated the ϕ nodes with a label information ℓ that ensures that the SSA syntax is self-contained and expressive enough to be equivalent to any imperative program syntax, as we show in the following. ϕ nodes that merge expressions declared at different loop depths are called $\mathsf{loop}_\ell\phi$ nodes and have a recursive semantics. $\mathsf{close}_\ell\phi$ nodes collect the values that come either from the loop ℓ or from before the loop ℓ, when the loop trip count is zero.

More traditional ϕ-nodes, also called "conditional-ϕ' in GCC, are absent from our core SSA syntax since they would only be required to handle conditional statements, which are absent from the syntax of IMP; these nodes would be handled by a proper combination of $\mathsf{loop}\phi$ and $\mathsf{close}\phi$ nodes [3].

[3]Note that we do not advocate a restructuring of the control flow graph, as from a purely technical point of view, a control flow restructuring operation would increase the size of the generated code, and thus harm the benefits of using the SSA representation.

The set of declarations[4] representing an SSA program is denoted in our framework as a finite function $\sigma \in \Sigma = Ide_{\mathsf{SSA}} \to SSA$ mapping each identifier to its defining expression.

Semantics. The semantics of an SSA expression $\mathcal{E}[\![]\!] \in SSA \to \Sigma \to N^m \to \mathcal{V}$ provides, given an SSA expression and an iteration space vector, its value. The semantics of an SSA program σ is thus a function with a finite domain, mapping identifiers I_h to the semantics of their values σI_h.

We give below the denotational semantics of an SSA program[5] tabulated by σ:

$$\begin{aligned}
\mathcal{E}[\![N]\!]\sigma k &= in_{\mathcal{V}}(N) \\
\mathcal{E}[\![I]\!]\sigma k &= \mathcal{E}[\![\sigma I]\!]\sigma k \\
\mathcal{E}[\![E_1 \oplus E_2]\!]\sigma k &= \mathcal{E}[\![E_1]\!]\sigma k \oplus \mathcal{E}[\![E_2]\!]\sigma k \\
\mathcal{E}[\![\mathsf{loop}_\ell\phi(E_1, E_2)]\!]\sigma k &= \begin{cases} \mathcal{E}[\![E_1]\!]\sigma k, & \text{if } k_\ell = 0, \\ \mathcal{E}[\![E_2]\!]\sigma k_{\ell-}, & \text{otherwise.} \end{cases} \\
\mathcal{E}[\![\mathsf{close}_\ell\phi(E_1, E_2)]\!]\sigma k &= \mathcal{E}[\![E_2]\!]\sigma k[\min\{x \mid \neg\mathcal{E}[\![E_1]\!]\sigma k[x/\ell]\}/\ell]
\end{aligned}$$

Constants such as N are denoted by themselves. As can be seen from the definition for identifiers I, we use the traditional syntactic "call-by-text" approach [Gor79] to handle the fixed point nature of an SSA program[6].

$\mathsf{loop}_\ell\phi$ expression, by their very iterative nature, are designed to represent the successive values of variables successively modified in imperative loop bodies. The first expression of a $\mathsf{loop}_\ell\phi$ represents the initial value while the second expression represents the expression inductively defined in the loop ℓ. The $\mathsf{loop}_\ell\phi$ expression represents an infinite stream of values, and it is the expression needed for the declaration of primitive recursive functions.

$\mathsf{close}_\ell\phi$ expression compute the final value of such inductive variables in loops guarded by a test expression E_1. The value of the $\mathsf{close}_\ell\phi$ expression is that of E_2 taken for the first iteration of the loop ℓ that makes E_1 false. Of course, when a loop is infinite, there is no iteration that exits the loop,

[4]There is no need to order the declarations, as in a declarative language the notion of sequence is inexistent. As we shall see in the IMP to SSA compiler, the information contained in statement sequence is completely transformed in the translation.

[5]Remember that k designates an iteration vector, k_ℓ the value contained at index ℓ, and $k_{\ell-}$ the decrement of the value at index ℓ.

[6]Strictly speaking, this denotational semantics is in fact an operational one since it doesn't use proper structural induction for identifiers (see [Sto77], p.338). A strictly denotational approach could have been defined as a semantical fixed point on a store mapping identifiers to values, but we found that the current formulation leads to a more intuitive wording of our main theorem and proof.

i.e., there is no k such that $\neg\mathcal{E}[\![E_1]\!]\sigma k$, and thus the set $\{x \mid \neg\mathcal{E}[\![E_1]\!]\sigma k[x/\ell]\}$ is empty. In such a case, $\min\emptyset$ corresponds to $\bot$. The $\mathsf{close}_\ell\phi$ expression provides the SSA the full power of the partial recursive functions.

Example. We informally illustrate in Figure 2.6 the semantics of SSA using an SSA program intended to be similar to the IMP program provided in Figure 2.5.

σ:

```
I1 → 7
J1 → 0
J2 → loop1φ(J1, J3)
J3 → J2 + I1
J4 → close1φ(J2 < 10, J2)
```

$\xrightarrow{\mathcal{E}[\![\sigma}$

$$\begin{aligned}
&\mathtt{I}_1 \to \lambda k.7\\
&\mathtt{J}_1 \to \lambda k.0\\
&\mathtt{J}_2 \to \lambda k.\begin{cases}\mathtt{J}_1(k) \text{ for } k = 0\\ \mathtt{J}_3(k-1) \text{ for } k > 0\end{cases}\\
&\mathtt{J}_3 \to \lambda k.\mathtt{J}_2(k) + \mathtt{I}_1(k) \text{ for } k \geq 0\\
&\mathtt{J}_4 \to \lambda k.14
\end{aligned}$$

Figure 2.6: Syntax and semantics of a loop-ϕ expression.

We use a different identifier (i.e., subscript) for each declared identifier (see for instance J) in the IMP program. Of course, all values are functions mapping iteration vectors to constants. To merge the two paths reaching in IMP the loop body, we use a $\mathsf{loop}\phi$ expression to combine the initial value of J and its successive iterated values within the loop. A $\mathsf{close}\phi$ expression "closes" the iterative function associated to $\mathtt{J}_2$ to retrieve its final value, obtained when the test expressions evaluates to *false*; in this case, this yields 14.

2.3.3 Conversion to SSA

We are now ready to specify how imperative constructs from IMP can be translated to SSA expressions. We use a denotational framework to specify formally this transformation process.

Specification. As any denotational specification, our transformation functions use states. These states $\theta = (\mu, \sigma) \in \mathcal{T} = M \times \Sigma$ have two components: $\mu \in M = Ide \to N^* \to Ide_{\mathsf{SSA}}$ maps imperative identifiers to SSA identifiers, yielding their latest SSA names (these can vary since a given identifier I can be used in more than one IMP assignment statement);

$\sigma \in \Sigma = Ide_{\mathsf{SSA}} \to SSA$ simply collects the SSA definitions associated to each identifier in the image of M.

The translation semantics $\mathcal{C}[\![]\!] \in Expr \to N^* \to M \to SSA$ for imperative expressions yields the SSA code corresponding to an imperative expression:

$$\begin{array}{rcl} \mathcal{C}[\![N]\!]h\mu & = & N \\ \mathcal{C}[\![I]\!]h\mu & = & R_{<h}(\mu I) \\ \mathcal{C}[\![E_1 \oplus E_2]\!]h\mu & = & \mathcal{C}[\![E_1]\!]h\mu \oplus \mathcal{C}[\![E_2]\!]h\mu \end{array}$$

As in the standard semantics for SSA, we need to find the reaching definition of identifiers, $R_{<h}$, although this time, since this is a compile-time translation process, we only look at the *syntactic* order corresponding to Dewey numbers.

The translation semantics of imperative statements $\mathcal{C}[\![]\!] \in Stmt \to N^* \to \mathcal{T} \to \mathcal{T}$ maps conversion states to updated conversion states. The cases for assignments and sequences are straightforward:

$$\begin{array}{rcl} \mathcal{C}[\![S_1; S_2]\!]h & = & \mathcal{C}[\![S_2]\!]h.2 \circ \mathcal{C}[\![S_1]\!]h.1 \\ \mathcal{C}[\![I = E]\!]h(\mu, \sigma) & = & (\mu[I_h/h/I], \sigma[\mathcal{C}[\![E]\!]h\mu/I_h]) \end{array}$$

since, for sequences, conversion states are simply propagated. For assignments, μ is extended by associating to the imperative identifier I the new SSA name I_h, to which the converted SSA right hand side expression is bound in σ, thus enriching the SSA program with a new binding for I_h.

As expected, most of the work is performed in while loops:

$$\begin{array}{l} \mathcal{C}[\![\mathsf{while}_\ell\ E\ \mathsf{do}\ S]\!]h(\mu, \sigma) = \theta_2 \text{ with} \\ \quad \theta_0 = (\mu[I_{h.0}/h.0/I]_{I \in Dom\ \mu}, \\ \qquad \sigma[\mathsf{loop}_\ell\phi(R_{<h}(\mu I), \bot)/I_{h.0}]_{I \in Dom\ \mu}), \\ \quad \theta_1 = \mathcal{C}[\![S]\!]h.1\theta_0, \\ \quad \theta_2 = (\mu_1[I_{h.2}/h.2/I]_{I \in Dom\ \mu_1}, \\ \qquad \sigma_1[\mathsf{loop}_\ell\phi(R_{<h}(\mu I), R_{<h.2}(\mu_1 I))/I_{h.0}] \\ \qquad\quad [\mathsf{close}_\ell\phi(\mathcal{C}[\![E]\!]h.1\mu_1, I_{h.0})/I_{h.2}]_{I \in Dom\ \mu_1}) \end{array}$$

where we note $\theta_i = (\mu_i, \sigma_i)$. We also used the notation $f[y/x]_{x \in S}$ to represent the extension of f to all values x in S with y.

As usual, the conversion process is, by induction, applied on the loop body S located at $h.1$. Yet, this cannot be performed in the original conversion state (μ, σ), since any imperative variable could be further modified in the loop body, creating a new binding which would be visible at the next iteration. To deal with this issue, a new Dewey number is introduced, $h.0$, preceding $h.1$, via which all variables[7] are bound to $\mathsf{loop}\phi$ nodes (note that only the SSA expressions corresponding to the control flow coming into the loop can be expressed at that point). It is now appropriate to convert the loop body in this updated conversion state; all references to variables will be to $\mathsf{loop}\phi$ nodes, as expected.

Similarly, after the converted loop body, a new Dewey number, $h.2$, following $h.1$, is introduced to bind all variables to $\mathsf{close}\phi$ nodes that represent their values when the loop exits (or $\perp$ if the loop is infinite, as we will see). All references to any identifier once the loop is performed are references to these $\mathsf{close}\phi$ expressions located at $h.2$, which follows, by definition of the lexicographic order on points, all other points present in the loop.

At this time, we are able to provide the entire definition for $\mathsf{loop}\phi$ expressions bound at level $h.0$, in particular the proper second subexpression within each $\mathsf{loop}\phi$ corresponds to the value of each identifier after one loop iteration.

Example. We find in Figure 2.7 the result of the conversion algorithm on our running example; as expected, this SSA program is the same code as the one in Figure 2.6, up to the renaming of the SSA identifiers. Note that all control-flow information has been removed from the IMP program, thus yielding a "pure", self-contained SSA form, without any need for additional, on-the-side control-flow data structure.

2.3.4 SSA Conversion Consistency

We are finally equipped with all the material required to express our main theorem. Our goal is to prove that our conversion process maintains the memory states consistency between the imperative and SSA representations. This relationship is expressed in the following definition:

[7] In fact, only the variables modified in the loop body need to be managed this way. We do not worry about such optimization here.

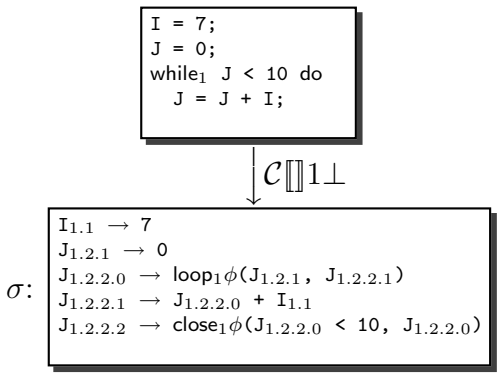


Figure 2.7: Conversion from IMP to SSA.

Definition 2.3.1 (Consistency) *A conversion state $\theta = (\mu, \sigma)$ is* consistent *with the memory state t at point $p = (h, k)$, noted $\mathcal{P}(\theta, t, p)$, iff*

$$\forall I \in Dom\ t, \mathcal{I}[\![I]\!]pt = \mathcal{E}[\![\mathcal{C}[\![I]\!]h\mu]\!]\sigma k$$

which specifies that, for any identifier, its value at a given point in the standard semantics is the same as its value in the SSA semantics when applied to its translated SSA equivalent (see Figure 2.8).

$$\begin{array}{ccc} Expr & \xrightarrow{\mathcal{C}[\![]\!]h\mu} & SSA \\ {\scriptstyle \mathcal{I}[\![]\!](h,k)t}\downarrow & & \downarrow{\scriptstyle \mathcal{E}[\![]\!]\sigma k} \\ v \in \mathcal{V} & = & v \in \mathcal{V} \end{array}$$

Figure 2.8: Consistency property $\mathcal{P}((\mu, \sigma), t, (h, k))$ for the translation of expressions.

This consistency requirement on identifiers can be straightforwardly extended to arbitrary expressions:

Lemma 2.3.1 (Consistency of Expression Conversion) *Given a consistent state $\mathcal{P}(\theta, t, p)$, and an expression $E \in Expr$,*

$$\mathcal{I}[\![E]\!]pt = \mathcal{E}[\![\mathcal{C}[\![E]\!]h\mu]\!]\sigma k$$

Proof. By induction on the structure of *Expr*:

- When $E = N$, trivial.

- When $E = I$, trivial.

- When $E = E_1 \oplus E_2$, by induction, applying the lemma to E_1 and E_2, both $\mathcal{I}[\![E_1]\!]pt = \mathcal{E}[\![\mathcal{C}[\![E_1]\!]h\mu]\!]\sigma k$ and $\mathcal{I}[\![E_2]\!]pt = \mathcal{E}[\![\mathcal{C}[\![E_2]\!]h\mu]\!]\sigma k$ hold.

$$\begin{aligned}
\mathcal{I}[\![E_1 \oplus E_2]\!]pt &= \mathcal{I}[\![E_1]\!]pt \oplus \mathcal{I}[\![E_2]\!]pt \\
&= \mathcal{E}[\![\mathcal{C}[\![E_1]\!]h\mu]\!]\sigma k \oplus \mathcal{E}[\![\mathcal{C}[\![E_2]\!]h\mu]\!]\sigma k \\
&= \mathcal{E}[\![\mathcal{C}[\![E_1]\!]h\mu \oplus \mathcal{C}[\![E_2]\!]h\mu]\!]\sigma k \\
&= \mathcal{E}[\![\mathcal{C}[\![E_1 \oplus E_2]\!]h\mu]\!]\sigma k
\end{aligned}$$

  So, the lemma also holds for $E_1 \oplus E_2$, finishing the proof. □

This directly leads to our main theorem, which proves the semantic correctness of the conversion process from imperative constructs to SSA expressions (as a shorthand, we note $p+ = (h, k)+ = (h+, k)$):

Theorem 2.3.2 (Consistency of Statement Conversion) *Given any statement S and for all θ, t, p that verify $\mathcal{P}(\theta, t, p)$, if $\theta' = \mathcal{C}[\![S]\!]h\theta$ and $t' = \mathcal{I}[\![S]\!]pt$, the property $\mathcal{P}(\theta', t', p+)$ holds.*

This theorem basically states that if the consistency property is satisfied for any point before a statement, then it is also verified for the statement that syntactically follows it.

Proof. By induction on the structure of *Stmt*, assuming $\mathcal{P}(\theta, t, p)$:

- for the assignment $[\![I = E]\!]$:

$$\begin{aligned}
(\mu', \sigma') &= \mathcal{C}[\![I = E]\!]h\theta = (\mu[I_h/h/I], \sigma[\mathcal{C}[\![E]\!]h\mu/I_h]), \\
t' &= \mathcal{I}[\![I = E]\!]pt = t[\mathcal{I}[\![E]\!]pt/p/I]
\end{aligned}$$

$$\begin{aligned}
\mathcal{I}[\![I]\!]p + t' &= R_{<p+}(t'I) && (\text{def } \mathcal{I}[\![]\!]) \\
&= \mathcal{I}[\![E]\!]pt && (\text{def } t') \\
&= \mathcal{E}[\![\mathcal{C}[\![E]\!]h\mu]\!]\sigma k && (\text{Lemma 2.3.1}) \\
&= \mathcal{E}[\![\mathcal{C}[\![E]\!]h\mu]\!]\sigma' k && (\text{extension of } \sigma') \\
&= \mathcal{E}[\![\sigma' I_h]\!]\sigma' k && (\text{def } \sigma') \\
&= \mathcal{E}[\![I_h]\!]\sigma' k && (\text{def } \mathcal{E}[\![]\!]) \\
&= \mathcal{E}[\![\mu' I h]\!]\sigma' k && (\text{def } \mu') \\
&= \mathcal{E}[\![R_{<h+}(\mu' I)]\!]\sigma' k && (\text{def } R_<) \\
&= \mathcal{E}[\![\mathcal{C}[\![I]\!]h + \mu']\!]\sigma' k && (\text{def } \mathcal{C}[\![]\!])
\end{aligned}$$

The extension to σ' is possible because it does not modify the reaching definitions: $R_{<p}$. So the property holds for I, but it also trivially holds for any $I' \neq I, I' \in Dom\ t$. So, $\mathcal{P}(\theta', t', p+)$ holds.

- for the sequence $[\![S_1; S_2]\!]$:

  Since there are no new bindings between h and $h.1$, $R_{<p} = R_{<(h.1,k)}$ and thus $\mathcal{P}(\theta, t, (h.1, k))$ holds.

  By induction, using the result of the theorem on S_1 with $\theta_1 = \mathcal{C}[\![S_1]\!]h.1\theta$ and $t_1 = \mathcal{I}[\![S_1]\!](h.1, k)t$, the property $\mathcal{P}(\theta_1, t_1, (h.1+, k))$ holds.

  Since $h.1+ = h.2$, by induction, using the result of the theorem on S_2, with $\theta_2 = \mathcal{C}[\![S_2]\!]h.2\theta_1$, and $t_2 = \mathcal{I}[\![S_2]\!](h.2, k)t_1$, then $\mathcal{P}(\theta_2, t_2, (h.2+, k))$ holds.

  So, the property $\mathcal{P}(\theta_2, t_2, p+)$ holds, since $h.2+ = h+$.

- for the loop $[\![\mathsf{while}_\ell\ E\ \mathsf{do}\ S]\!]$:

  The recursive semantics for while loops suggests to use fix point induction ([Sto77], p.213), but this would require us to define new properties and functionals operating on (θ, t, p) as a whole while changing the definition of P to handle ordinals. We prefer to keep a simpler profile here, and give a somewhat ad-hoc but more intuitive proof.

  We will need a couple of lemmas to help us build the proof. As a shorthand, we note $\theta_{ij} = (\mu_i, \sigma_j)$.

  Lemma 2.3.3 *With $t = t_0$, $\mathcal{P}_0 = \mathcal{P}(\theta_{12}, t_0, h.1, k[0/\ell])$ holds.*

  This lemma states that if $\mathcal{P}$ is true at loop entry, then it remains true just before the loop body of the first iteration, at point $(h.1, k[0/\ell])$.

PROOF. $\forall I \in Dom\ t$:

$$
\begin{aligned}
&\mathcal{I}[\![I]\!](h.1, k[0/\ell])t = \\
&= R_{<(h.1,k[0/\ell])}(tI) && (\text{def } \mathcal{I}[\![]\!]) \\
&= R_{<p}(tI) && (\text{def } t) \\
&= \mathcal{I}[\![I]\!]pt && (\text{def } \mathcal{I}[\![]\!]) \\
&= \mathcal{E}[\![\mathcal{C}[\![I]\!]h\mu]\!]\sigma k && (\mathcal{P}(\theta, t, p)) \\
&= \mathcal{E}[\![R_{<h}(\mu I)]\!]\sigma k && (\text{def } \mathcal{C}[\![]\!]) \\
&= \mathcal{E}[\![R_{<h}(\mu I)]\!]\sigma k[0/\ell] && (\text{first iteration}) \\
&= \mathcal{E}[\![R_{<h}(\mu I)]\!]\sigma_0 k[0/\ell] && (\text{extension to } \sigma_0) \\
&= \mathcal{E}[\![\mathsf{loop}_\ell\phi(R_{<h}(\mu I), \bot)]\!]\sigma_0 k[0/\ell] && (\text{def } \mathsf{loop}_\ell\phi) \\
&= \mathcal{E}[\![\sigma_0 I_{h.0}]\!]\sigma_0 k[0/\ell] && (\text{def } \sigma_0) \\
&= \mathcal{E}[\![I_{h.0}]\!]\sigma_0 k[0/\ell] && (\text{def } \mathcal{E}[\![]\!]) \\
&= \mathcal{E}[\![\mu_0 Ih.0]\!]\sigma_0 k[0/\ell] && (\text{def } \mu_0) \\
&= \mathcal{E}[\![R_{<h.1}(\mu_0 I)]\!]\sigma_0 k[0/\ell] && (\text{def } R_<) \\
&= \mathcal{E}[\![\mathcal{C}[\![I]\!]h.1\mu_0]\!]\sigma_0 k[0/\ell] && (\text{def } \mathcal{C}[\![]\!])
\end{aligned}
$$

So, $\mathcal{P}(\theta_0, t, h.1, k[0/\ell])$ holds. The extension of θ_0 to θ_{12} concludes the proof of the Lemma 2.3.3. □

Lemma 2.3.4 *Given* $\mathcal{P}_{x-1} = \mathcal{P}(\theta_{12}, t_{x-1}, (h.1, k[x-1/\ell]))$ *for some* $x \geq 1$, $\mathcal{P}_x = \mathcal{P}(\theta_{12}, t_x, (h.1, k[x/\ell]))$ *holds.*

This second lemma ensures that if $\mathcal{P}$ is true at iteration $x-1$, then it stays the same at iteration x. Note that the issue of whether we will indeed enter the loop again or exit it altogether is no factor here.

PROOF. Let $t_x = \mathcal{I}[\![S]\!](h.1, k[x-1/\ell])t_{x-1}$. By induction, applying the theorem to S, we know that the property

$$\mathcal{P}'_{x-1} = \mathcal{P}(\theta_{12}, t_x, (h.2, k[x-1/\ell]))$$

holds, since $h.1+ = h.2$, and $\theta_{12} = \mathcal{C}[\![S]\!]h.1\theta_{12}$, as $\mathcal{C}[\![]\!]$ is idempotent.

$$
\begin{aligned}
&\mathcal{I}[\![I]\!](h.1, k[x/\ell])t_x = \\
&= R_{<(h.1,k[x/\ell])}(t_x I) && (\text{def } \mathcal{I}[\![]\!]) \\
&= R_{<(h.2,k[x-1/\ell])}(t_x I) && (\text{def } R_<) \\
&= \mathcal{I}[\![I]\!](h.2, k[x-1/\ell])t_x && (\text{def } \mathcal{I}[\![]\!]) \\
&= \mathcal{E}[\![\mathcal{C}[\![I]\!]h.2\mu_1]\!]\sigma_2 k[x-1/\ell] && (\mathcal{P}'_{x-1}) \\
&= \mathcal{E}[\![R_{<h.2}(\mu_1 I)]\!]\sigma_2 k[x-1/\ell] && (\text{def } \mathcal{C}[\![]\!]) \\
&= \mathcal{E}[\![\mathsf{loop}_\ell\phi(R_{<h}(\mu I), R_{<h.2}(\mu_1 I))]\!]\sigma_2 k[x/\ell] && (\text{def } \mathsf{loop}_\ell\phi) \\
&= \mathcal{E}[\![\sigma_2 I_{h.0}]\!]\sigma_2 k[x/\ell] && (\text{def } \sigma_2) \\
&= \mathcal{E}[\![I_{h.0}]\!]\sigma_2 k[x/\ell] && (\text{def } \mathcal{E}[\![]\!]) \\
&= \mathcal{E}[\![\mu_1 I h.0]\!]\sigma_2 k[x/\ell] && (\text{def } \mu_1) \\
&= \mathcal{E}[\![R_{<h.1}(\mu_1 I)]\!]\sigma_2 k[x/\ell] && (\text{def } R_<) \\
&= \mathcal{E}[\![\mathcal{C}[\![I]\!]h.1\mu_1]\!]\sigma_2 k[x/\ell] && (\text{def } \mathcal{C}[\![]\!])
\end{aligned}
$$

This concludes the proof of Lemma 2.3.4. □

We are now ready to tackle the different cases that can occur during evaluation. These three cases are:

1. when the loop is not executed, that is when the exit condition is false before entering the loop body: $\neg\mathcal{I}[\![E]\!]t(h.1, k[0/\ell])$. Based on Lemma 2.3.3, $\mathcal{P}(\theta', t', p+)$ holds, as $\theta' = \theta_2$ that extends θ_{12},

$t = t'$ as defined by the exit of the while_ℓ in $\mathcal{I}[\![]\!]$, and $k[0/\ell] = k$:

$$
\begin{aligned}
&\mathcal{I}[\![I]\!](p+)t = \\
&= R_{<p+}(tI) && (\text{def } \mathcal{I}[\![]\!]) \\
&= R_{<p}(tI) && (\text{def } t) \\
&= \mathcal{I}[\![I]\!]pt && (\text{def } \mathcal{I}[\![]\!]) \\
&= \mathcal{E}[\![\mathcal{C}[\![I]\!]h\mu]\!]\sigma k && (\mathcal{P}) \\
&= \mathcal{E}[\![R_{<h}(\mu I)]\!]\sigma k && (\text{def } \mathcal{C}[\![]\!]) \\
&= \mathcal{E}[\![R_{<h}(\mu I)]\!]\sigma_2 k && (\text{extension to } \sigma_2) \\
&= \mathcal{E}[\![\mathsf{loop}_\ell\phi(R_{<h}(\mu I), R_{<h.2}(\mu_1 I))]\!]\sigma_2 k && (\text{def } \mathsf{loop}_\ell\phi) \\
&= \mathcal{E}[\![\sigma_2 I_{h.0}]\!]\sigma_2 k && (\text{def } \sigma_2) \\
&= \mathcal{E}[\![I_{h.0}]\!]\sigma_2 k && (\text{def } \mathcal{E}[\![]\!]) \\
&= \mathcal{E}[\![\mathsf{close}_\ell\phi(\mathcal{C}[\![E]\!]h.1\mu_1, I_{h.0})]\!]\sigma_2 k && (\text{def } \mathsf{close}_\ell\phi) \\
&= \mathcal{E}[\![\sigma_2 I_{h.2}]\!]\sigma_2 k && (\text{def } \sigma_2) \\
&= \mathcal{E}[\![I_{h.2}]\!]\sigma_2 k && (\text{def } \mathcal{E}[\![]\!]) \\
&= \mathcal{E}[\![\mu_2 I h.2]\!]\sigma_2 k && (\text{def } \mu_2) \\
&= \mathcal{E}[\![R_{<h+}(\mu_2 I)]\!]\sigma_2 k && (\text{def } R_<) \\
&= \mathcal{E}[\![\mathcal{C}[\![I]\!]h + \mu_2]\!]\sigma_2 k && (\text{def } \mathcal{C}[\![]\!])
\end{aligned}
$$

2. when the loop is executed a finite number of times, that is when the loop body is executed at least once: let $\omega > 0$ be the first iteration on which the loop condition becomes false:

$$
\begin{aligned}
\omega &= \min\{x \mid \neg\mathcal{I}[\![E]\!]t_\omega(h.1, k[x/\ell])\} \\
&= \min\{x \mid \neg\mathcal{E}[\![\mathcal{C}[\![E]\!]h.1\mu_1]\!]\sigma_2 k[x/\ell]\} && (\text{Lemma 2.3.1})
\end{aligned}
$$

By Lemma 2.3.3 and 2.3.4, $\mathcal{P}'_\omega = \mathcal{P}(\theta_{12}, t_\omega, h.2, k[\omega - 1/\ell])$ holds,

and then, $\mathcal{P}(\theta', t_\omega, p+)$ holds:

$$
\begin{array}{ll}
\mathcal{I}[\![I]\!](p+)t_\omega = & \\
= R_{<p+}(t_\omega I) & (\text{def } \mathcal{I}[\![]\!]) \\
= R_{<(h.2,k_\omega)}(t_\omega I) & (\text{def } R_<) \\
= \mathcal{I}[\![I]\!](h.2, k_\omega)t_\omega & (\text{def } \mathcal{I}[\![]\!]) \\
= \mathcal{E}[\![\mathcal{C}[\![I]\!]h.2\mu_1]\!]\sigma_2 k[\omega - 1/\ell] & (\mathcal{P}'_\omega) \\
= \mathcal{E}[\![R_{<h.2}(\mu_1 I)]\!]\sigma_2 k[\omega - 1/\ell] & (\text{def } \mathcal{C}[\![]\!]) \\
= \mathcal{E}[\![\mathsf{loop}_\ell\phi(R_{<h}(\mu I), R_{<h.2}(\mu_1 I))]\!]\sigma_2 k_\omega & (\text{def } \mathsf{loop}_\ell\phi) \\
= \mathcal{E}[\![\sigma_2 I_{h.0}]\!]\sigma_2 k_\omega & (\text{def } \sigma_2) \\
= \mathcal{E}[\![I_{h.0}]\!]\sigma_2 k_\omega & (\text{def } \mathcal{E}[\![]\!]) \\
= \mathcal{E}[\![\mathsf{close}_\ell\phi(\mathcal{C}[\![E]\!]h.1\mu_1, I_{h.0})]\!]\sigma_2 k & (\text{def } \mathsf{close}_\ell\phi) \\
= \mathcal{E}[\![\sigma_2 I_{h.2}]\!]\sigma_2 k & (\text{def } \sigma_2) \\
= \mathcal{E}[\![I_{h.2}]\!]\sigma_2 k & (\text{def } \mathcal{E}[\![]\!]) \\
= \mathcal{E}[\![\mu_2 I h.2]\!]\sigma_2 k & (\text{def } \mu_2) \\
= \mathcal{E}[\![R_{<h+}(\mu_2 I)]\!]\sigma_2 k & (\text{def } R_<) \\
= \mathcal{E}[\![\mathcal{C}[\![I]\!]h + \mu_2]\!]\sigma_2 k & (\text{def } \mathcal{C}[\![]\!])
\end{array}
$$

Finally, using Kleene's Fixed Point Theorem [Sto77], we can relate the least fixed point $\text{fix}(W)$ used to define the standard semantics of while loops and the successive iterations $W^i(\bot)$ of the loop body:

$$
\begin{array}{rcl}
t' & = & \text{fix}(W)(h, k[0/\ell])t \\
& = & \lim\limits_{i\to\infty} W^i(\bot)(h, k[0/\ell])t \\
& = & W^\omega(\bot)(h, k[0/\ell])t \\
& = & t_\omega
\end{array}
$$

and so $\mathcal{P}(\theta', t', p+)$ holds.

3. when the loop is infinite: $t' = \lim_{i\to\infty} W^i(\bot)(h, k[0/\ell])t = \bot$. Thus:

$$
\begin{aligned}
&\mathcal{I}[\![I]\!](p+)\bot = \bot = && (\text{def } \mathcal{I}[\![]\!]) \\
&= \mathcal{E}[\![\mathsf{close}_\ell\phi(\mathcal{C}[\![E]\!]h.1\mu_1, I_{h.0})]\!]\sigma_2 k && (\min\emptyset = \bot) \\
&= \mathcal{E}[\![\sigma_2 I_{h.2}]\!]\sigma_2 k && (\text{def } \sigma_2) \\
&= \mathcal{E}[\![I_{h.2}]\!]\sigma_2 k && (\text{def } \mathcal{E}[\![]\!]) \\
&= \mathcal{E}[\![\mu_2 Ih.2]\!]\sigma_2 k && (\text{def } \mu_2) \\
&= \mathcal{E}[\![R_{<h+}(\mu_2 I)]\!]\sigma_2 k && (\text{def } R_<) \\
&= \mathcal{E}[\![\mathcal{C}[\![I]\!]h + \mu_2]\!]\sigma_2 k && (\text{def } \mathcal{C}[\![]\!])
\end{aligned}
$$

So, $\mathcal{P}(\theta', t', p+)$ holds.

thus completing the proof of our main theorem, and ensuring the consistency of the whole SSA conversion process. □

We are left with the simple issue of checking that state consistency is satisfied for the initial states.

Lemma 2.3.5 *$\mathcal{P}(\bot, \bot, (1, 0^m))$ holds.*

Proof.

$$
\begin{aligned}
&\mathcal{I}[\![I]\!](1, 0^m)\bot = \\
&= R_{<1.1}(\bot I) && (\text{def } \mathcal{I}[\![]\!]) \\
&= \bot \\
&= \mathcal{E}[\![R_{<1}(\bot I)]\!]\bot 0^m && (\text{def } R_<) \\
&= \mathcal{E}[\![\mathcal{C}[\![I]\!]1\bot]\!]\bot 0^m && (\text{def } \mathcal{C}[\![]\!])
\end{aligned}
$$

□

The final theorem wraps things up by showing that after evaluating an SSA-converted program from consistent initial states, we end up in states that remain consistent. Note that this remains true even if the whole program loops.

Theorem 2.3.6 *Given $S \in Stmt$, with $\theta = \mathcal{C}[\![S]\!]1\bot$, and $t = \mathcal{I}[\![S]\!](1, 0^m)\bot$, the property $\mathcal{P}(\theta, t, (2, 0^m))$ holds.*

Proof. Trivial using Lemma 2.3.5 and Theorem 2.3.2. □

2.3.5 Discussion

Even though the initial purpose of our work is to provide a firm foundation to the use of SSA in modern compilers, our results also yield an interesting theoretical insight on the computational power of SSA.

Recursive Partial Functions Theory. The mathematical wording of the Consistency Property 2.3.1 underlines a key aspect of the SSA conversion process. While p only occurs on the left hand side of the consistency equality, the syntactic location h and the iteration space vector k are uncoupled in the right-hand side expression. Thus, via the SSA conversion process, the standard semantics gets staged, informally getting "curryied" from $Stmt \rightarrow (N^* \times N^m) \rightarrow T \rightarrow T$ to $Stmt \rightarrow N^* \rightarrow N^m \rightarrow T \rightarrow T$; this is also visible on Figure 2.8 where the pair (h, k) is used on the left arrow, while h and k occur separately on the top and on the right arrows. This perspective change is rather profound, since it uncouples syntactic sequencing from run time iteration space sequencing.

There exists a formal computing model that is particularly well suited to describing iteration behaviors, namely Kleene's theory of partial recursive functions [Sto77]. In fact, the SSA appears to be a syntactic variant of such a formalism. We provide below a rewriting of SSA bindings to recursive function definitions.

First, to each SSA identifier I, we associate a function $I(k)$, and translate any SSA expression involving neither loopϕ nor closeϕ nodes[8] as function calls:

$$\begin{aligned} \mathcal{K}[\![N]\!]k &= N \\ \mathcal{K}[\![I]\!]k &= I(k) \\ \mathcal{K}[\![E_1 \oplus E_2]\!]k &= \oplus(\mathcal{E}[\![E_1]\!]k, \mathcal{E}[\![E_2]\!]k) \end{aligned}$$

Then, to collect partial recursive function definitions corresponding to an SSA program σ, we simply gather all the definitions for each binding, $\bigcup_{I \in Dom\ \sigma} \mathcal{K}[\![I, \sigma I]\!]$. Basically, for loop$\phi$ expressions, we simply rewrite the two cases corresponding to their standard semantics. For closeϕ expressions, we add an ancillary function that computes the minimum value (if any) of the loop counter corresponding to the number of iterations required to compute

[8] Without loss of generality, we assume that ϕ nodes only occur as top-level expression constructors.

the final value, and plug it into the final expression. This is formally defined as follows, using $k_{p,q}$ as a shorthand for $k_p, k_{p+1}, ..., k_{q-1}, k_q$:

$$
\begin{aligned}
&\mathcal{K}[\![I, \mathsf{loop}_\ell \phi(E_1, E_2)]\!]k = \\
&\quad \{I(k_{1,\ell-1}, 0, k_{\ell+1,m}) = \mathcal{K}[\![E_1]\!](k_{1,\ell-1}, 0, k_{\ell+1,m}), \\
&\qquad I(k_{1,\ell-1}, x+1, k_{\ell+1,m}) = \mathcal{K}[\![E_2]\!](k_{1,\ell-1}, x, k_{\ell+1,m})\} \\
&\mathcal{K}[\![I, \mathsf{close}_\ell \phi(E_1, E_2)]\!]k = \\
&\quad \{\mathtt{min}_I(k_{1,\ell-1}, k_{\ell+1,m}) = \\
&\qquad\qquad (\mu y. \mathcal{K}[\![E_1]\!](k_{1,\ell-1}, y, k_{\ell+1,m}) = 0), \\
&\qquad I(k) = \mathcal{K}[\![E_2]\!](k_{1,\ell-1}, \mathtt{min}_I(k_{1,\ell-1}, k_{\ell+1,m}), k_{\ell+1,m})\} \\
&\mathcal{K}[\![I, E]\!]k = \{I(k) = \mathcal{K}[\![E]\!]k\}
\end{aligned}
$$

where μ is Kleene's minimization operator. We also assumed that boolean values are coded as integers (*false* is 0).

Example. As an example of this transformation to partial recursive functions, we provide below the translation of our running example (see Figure 2.9) into partial recursive functions. For increased readability, we renamed variables to use shorter indices.

$$
\begin{aligned}
&\mathtt{I}_1(k_1) = 7 \\
&\mathtt{J}_1(k_1) = 0 \\
&\mathtt{J}_2(0) = \mathtt{J}_1(0) \\
&\mathtt{J}_2(x+1) = \mathtt{J}_3(x) \\
&\mathtt{J}_3(k_1) = +(\mathtt{J}_2(k_1), \mathtt{I}_1(k_1)) \\
&\mathtt{min}_{\mathtt{J}_4}() = (\mu y. < (\mathtt{J}_2(y), 10) = 0) \\
&\mathtt{J}_4(k_1) = \mathtt{J}_2(\mathtt{min}_{\mathtt{J}_4}())
\end{aligned}
$$

Figure 2.9: Partial recursive functions example.

Our conversion process from IMP to SSA can thus be seen as a way of converting any Random Access Machine (RAM) program [Jon97] to a set of Kleene's partial recursive functions, thus providing a new proof of Turing's Equivalence Theorem between these two computational models, previously typically proven using simulation.

Assignments versus SSA. Based on the denotational semantics of the SSA it is possible to see that the SSA is a declarative language: a language that is radically different compared to the imperative languages. A strange consonance occurs when considering the original name of the SSA, the Static Single Assignment, in the perspective of the definition of the SSA that we have seen in this chapter: indeed, declarative languages do not contain assignments, and this sounds even stranger if we consider the definition and the purpose of assignments that modern languages have inherited from the early programming languages, as described by John Backus in the Turing Award Lecture of 1977 [Bac78]:

> *[. . .],*
> *the assignment statement splits programming into two worlds. The first world comprises the right sides of assignment statements. This is an orderly world of expressions, a world that has useful algebraic properties (except that those properties are often destroyed by side effects). It is the world in which most useful computation takes place.*
>
> *The second world of conventional programming languages is the world of statements. The primary statement in that world is the assignment statement itself. All the other statements of the language exist in order to make it possible to perform a computation that must be based on this primitive construct: the assignment statement.*
>
> *This world of statements is a disorderly one, with few useful mathematical properties. Structured programming can be seen as a modest effort to introduce some order into this chaotic world, but it accomplishes little in attacking the fundamental problems created by the word-at-a-time von Neumann style of programming, with its primitive use of loops, subscripts, and branching flow of control.*

This chapter provides a paradigmatic shift for the SSA language: it is no longer possible to speak about assignments in the SSA form, as assignments are part of a purely imperative programming language. We will continue to use the SSA name for this language even if this name is not appropriate. The SSA is a first step into abstracting the imperative language semantics under the form of a higher level language: a language of declarations. In this language, the notion of store disappears, replaced by the notion of declarations and stream of values.

2.3.6 Future Work

We only looked at the IMP- to-SSA conversion process. A natural dual problem of course arises, namely the so-called "out-of-SSA" [CFR+91, BCHS98, SJGS99] issue: a way of pretty-printing SSA programs using imperative-like programming language syntax such as IMP. This is of utmost importance when one considers for instance the issues of debugging or code generation. In GCC, this is dealt with using a graph algorithm [GCC] operating on the control-flow data structure decorated with the SSA annotations used in its middle end.

For our approach, this technique could also be used in a similar fashion, assuming we kept around the control-flow graph from which our SSA code has been generated. A more intriguing question is whether such an out-of-SSA IMP code generator could be designed using only our self-contained SSA syntax. In a perfect world, one would indeed want to get back the original IMP code from which SSA has been generated. This requires reconstructing the while loop structure using data dependence within SSA code, together with an intelligent ordering of code generation for each binding in σ to minimize code duplication.

2.4 Conclusion

We successively have seen several descriptions of the SSA: the original practical description of the construction of the SSA, and then other formalizing attempts. The main contribution described in this chapter is the denotational semantics of the SSA together with the translation of a minimal imperative language to the SSA and the consistency proof for this translation. We then have seen some of the implications of this formal definition: the impact of the translation algorithm on the recursive partial functions, and the insight on the classification of the SSA language in the family of declarative languages.

Although the SSA compiler technology was born in imperative language compilers from the need of abstracting the store semantics of the von Neumann programming style, the SSA has the potential to unify several programming paradigms, and provide a common compiler infrastructure for translating languages to von Neumann architectures.

After this very high level presentation of the SSA, we will consider more pragmatic aspects of program analysis: the next chapter presents the construction of static analyzers on the SSA form, for extracting interesting constructs that are simple enough to be handled in several compiler techniques.

Chapter 3

From SSA to Abstract SSA in PROLOG

One of the most interesting practical aspects of the SSA representation comes from the observation that data flow problems are described atomically: in the classical data flow analysis for imperative programs, the analysis has to be performed on all the scalar variables modified in the program, whereas once the imperative program has been translated to SSA form, the resolution of the same data flow problem will determine the solution based only on a subset of the variables modified in the program, on which there is a data dependence. The SSA automatically clusters the declarations following the scalar data dependences. When all the scalar variables involved in such a cluster have statically known initial values, it sometimes is possible to give precise values to each of the variables members of that cluster. However, when some of the initial conditions are unknown, it is still possible to characterize the cluster's members, but only using some approximation: this chapter introduces the notion of Abstract SSA, or ASSA, obtained from SSA by an approximation. For obtaining the ASSA, I have proposed two versions of an algorithm that translates programs written in SSA to ASSA. The first algorithm will use PROLOG for representing the source and target representations [PCJS06b], and the translation algorithm uses the unification engine of PROLOG. The next chapter will describe a second algorithm [PCS05] using a more informal imperative language, that is more intuitive for engineers that would like to adapt this algorithm in their compiler infrastructure.

3.1 ASSA: Abstract SSA

After having seen, in the last chapter, an exact semantics of the SSA language, we would like to use this precise description of the semantics of the SSA language for building static analyzers. We will see in this section a survey of the existing tools used in static analysis and will propose the Abstract SSA form, a representation that can practically be used for static analyses, and for avoiding static uncomputable properties.

Static Analysis. In opposition to code instrumentation, testing or profiling, the static analysis is performed without knowing the context in which the program will be executed. Thus, static analysis obtains more general solutions, but most of the interesting questions become uncomputable: every property that is not true or false for all the programs is undecidable, a result commonly known under the name of Rice's theorem [Rog87, Jon97]. In order to avoid uncomputable properties, a static analyzer uses safe approximations of the behavior of the program, a trade between the cost of static computation and the precision of the provided answers: in some cases the abstract representations would not contain enough information for allowing the static analyzer to output a definitive answer, in which case the result is an uncertain answer, "don't know".

Concrete and Abstract Semantics. The precise semantics of the program is also named the concrete semantics. Approximations of the concrete semantics lead to abstract semantics: a part of the information encoded in the concrete semantics is lost in a process that can guarantee some safety properties. In general the concrete and the abstract elements are encoded using some ordered set, or domain, and the approximation operation, or abstraction, is a total function mapping elements from the concrete domain to elements of the abstract domain. A safe abstraction function conserves informational order properties in the starting and target domains.

Theories of Abstraction. The theory of abstraction has independently evolved in the abstract interpretation and artificial intelligence domains as practical methods for dealing with either large amounts of data, or for representing uncertainty in static analyses of programs. In practice, abstractions are defined by a mapping between two sets that preserves some properties and that reduces the complexity.

Definition 3.1.1 (Abstraction) *An abstraction is a triplet* $(\Sigma_1, \Sigma_2, \alpha)$*, with* Σ_1 *and* Σ_2 *two sets with* $\Sigma_2 \subseteq \Sigma_1$*, and a total function* $\alpha : \Sigma_1 \to \Sigma_2$

called the abstraction function, that maps the elements of Σ_1 *onto that of* Σ_2.

The usefulness of abstractions arises from the fact that it sometimes is simpler to work on the abstract set, Σ_2, that potentially contains fewer elements than Σ_1, and on which some properties may be decidable. Based on the results of the computation in the abstract set, it is then possible to infer an over-approximation for the result of the computation on Σ_1 using a concretization function.

Definition 3.1.2 (Concretization) *Let* $(\Sigma_1, \Sigma_2, \alpha)$ *be an abstraction. A function* $\gamma : \Sigma_2 \to \mathcal{P}(\Sigma_1)$ *is called a concretization function. It maps elements from* Σ_2 *to sets of elements of* Σ_1.

Abstract Interpretation. The semantics of a programming language can be more or less precise, depending on the degree of detail captured by the description of program execution. The framework of abstract interpretation [CC77, CC79] can be used to automatically define several semantics layers, corresponding to different levels of precision. Each semantics layer is represented by a partially ordered set, or domain. The relation between two semantics layers is defined using an approximation translation, which maps elements of the concrete domain to elements of the abstract domain. In order to guarantee the conservation of concrete semantics properties, the approximation and concretization functions have to satisfy the following properties, for some given elements from the concrete domain $c_1, c_2 \in \Sigma_1$, and for some elements of the abstract domain $a_1, a_2 \in \Sigma_2$:

- α monotone: $c_1 \subseteq c_2 \Rightarrow (\alpha(c_1) \subseteq \alpha(c_2))$,
- γ monotone: $a_1 \subseteq a_2 \Rightarrow (\gamma(a_1) \subseteq \gamma(a_2))$,
- $\gamma \circ \alpha$ extensive: $c_1 \subseteq \gamma(\alpha(c_1))$,
- $\alpha \circ \gamma$ reductive: $\alpha(\gamma(a_1)) \subseteq a_1$.

A coarse approximation of a concrete domain leads to a less precise abstract domain on which the semantics can be practically computable. It is sometimes the case that the operations on the abstract domain are exact, i.e. computing on the abstract set produces exactly the same result, without loss of precision, as in the concrete set. In this case the order is trivially preserved, as the abstraction and concretization functions are bijections on some subsets of the abstract and concrete sets.

Theory of Abstract Domains. As we have seen, approximation functions map elements from a concrete semantics domain to an abstract domain. Once an abstract domain is formalized, it is possible to use the abstract interpretation framework for automatically defining the static analyzer corresponding to the translation of the concrete semantics domain to the abstract domain. The importance of abstract domains in the framework of abstract interpretation is illustrated by the number of abstract domains that have been formalized:

- signs,
- intervals [Moo66],
- the polyhedral domain [CH78],
- the simple and linear congruence domains [Gra91, Cla96b, Mei04],
- the trapezoidal linear congruence domains [Mas92, Mas93],
- the octagon domain [Min01],
- the ellipsoid domain [Fer04, Fer05].

All these abstract domains can naturally be used in the framework of abstract interpretation for representing approximated semantics. We now will see how to use this abstract interpretation framework to represent approximations of run-time behaviors of programs in SSA form.

Abstractions from SSA Programs. By representing parts of the exact semantics of SSA programs with elements belonging to one of the abstract domains presented above, one can obtain several abstraction views of the behavior of the program at run-time. This SSA representation with declarations containing elements in an abstract domain will be called the Abstract SSA, or ASSA, and we will describe several algorithms that use this representation for statically computing properties of the program, and for building other static analyses.

The selection of the proper abstraction level is a difficult task, and cannot be performed automatically: it depends on the precision of the information required by a computation. For example, the value range propagation [Pat95] is able to infer useful information from integer intervals, whereas such information would probably be useless in the computation of the exact loop trip count, that would require a more precise information under the form of a symbolic expression. For this reason, it is impossible to provide a unique,

perfect abstraction view, in terms of computability and of information precision, for all the users of the information extracted by a static analyzer. Instead, the interface to the static analyzer can provide a mechanism for filtering the representation through several abstraction functions, allowing the users of this information to select an appropriate abstraction level. As we will see in much more details in Section 4.4, we will use the notion of instantiation of SSA declarations in some abstract domain for practically implementing such a mechanism.

In the following we consecutively will see two different, practical ways to work with abstract domains instead of the precise semantics of the SSA: first we will use a logic programming language for describing SSA programs, and for defining abstract SSA semantics, then we will use an imperative language for describing the same static analyzers, and finally we will see how I integrated these static analyzers in a real compiler, the GNU Compiler Collection: GCC.

3.2 SSA to ASSA in Logic Programming

After a short description of the specificities of logic programming and PROLOG, we will see how to represent SSA programs in PROLOG, and how to handle SSA programs containing abstract elements. Finally we will see a convenient representation for self declarations that will provide a compressed non recursive form of the SSA declarations.

3.2.1 Logic Programming and PROLOG

Logic programming was first proposed by John McCarthy in 1958 as a means of computing using mathematical logic: programs are defined by a tuple containing a set of assertions or declarations, and a query that has to be satisfied in this base of facts. One of the most popular logic programming languages, PROLOG, was created by Alain Colmerauer and Robert Kowalski in 1972, and later standardized, in 1995, by the International Organization for Standardization [ISO95]. For all the following experiments, we will use standard PROLOG. In particular, I have used a free implementation of PROLOG, the GNU PROLOG [GNU]. In the following, we will use its syntax, as well as its unification and Horn clause resolution semantics.

GNU PROLOG. The GNU PROLOG is a compiler written by Daniel Diaz. As many other PROLOG compilers, the GNU PROLOG uses an intermediate representation commonly called the Warren Abstract Machine, or WAM

[AK99]. The assembly code destined to be transformed into an executable for the target machine is then generated from the WAM code. The WAM model contains a memory architecture and an instruction set suited to translate PROLOG programs to common von Neumann computer architectures.

Syntax of PROLOG Programs. A PROLOG program is defined by a set of declarations composed of first order terms, that are composed of:

- variables denoted by a capitalized identifier, or by the "_" syntactic element that designates an anonymous variable,
- constants denoted by an identifier starting with a lower-case letter,
- and functors of the form $f(t_1, \ldots, t_n)$, with $t_1, \ldots, t_n$ first order terms, n is called the arity of the functor. The functor is also simply referred to as $f(n)$, as functors with different arity are not equal. Constants are special cases of functors with arity 0.

A predicate is syntactically defined with the " :- " construct, that can be read "if", and that separates the left hand side predicate to be defined from the fact that has to be satisfied on its right hand side. Facts are defined by conjunctions and disjunctions of first order terms. Syntactically, conjunctions are represented by a comma ", ", and disjunctions are represented by a semi-column ";". A fact ends with a point ".". A PROLOG program is constituted of a list of facts evaluated in the sequential order in which they appear in the text.

Semantics of PROLOG Programs. A query is a first order term that is not a variable. The semantics of a tuple "program-query" is the most general unifier of the program and the query. The operational semantics of a PROLOG program is given by some unification algorithm, like the linear unification algorithm presented in [PW78].

Abstractions as Herbrand Universes. A logic framework for abstractions can be described in terms of Herbrand universes: the concrete set can be defined as the set of all ground terms constructed from functors and constants; the abstraction mapping can be defined as a functor transforming every ground term from the starting set to other ground terms composed of constants and functors in the abstract set.

PROLOG as Prototyping Language. As we will see in this section, the tools provided by PROLOG suit the needs of prototyping SSA transformations: the execution of PROLOG programs are query driven similarly to the query driven access to the definitions of scalar variables in the SSA representation. PROLOG's unification engine provides a natural, practical technique to describe complex algorithms that transform the SSA language. The complex description of the algorithms that we have described in [PCS05], and that we will see in Section 4.4, is mainly due to the description of a unification algorithm similar to the algorithm presented in [PW78].

3.2.2 Representing SSA in PROLOG

We will see a possible encoding of the SSA language using PROLOG: relations between scalar variables are described with PROLOG predicates. Then we will specify a small interpreter for SSA expressions on top of the unification engine of PROLOG, that will provide an exact meaning to each PROLOG predicate.

Definition of SSA Predicates. An SSA program tabulated by a relation σ, defined as in the previous chapter[1], can be represented in PROLOG using a binary predicate: `sig(2)`. The SSA expressions are then composed of names declared in the σ mapping, boolean and integer variables, and a mix of classical binary operators $+$, $-$, $\leq$, $\wedge$, and $\neg$. As in the previous chapter, conditional phi expressions will not be represented, as theoretically they do not bring any expressiveness and can be represented by loops. Predicates `lphi(3)`, and `cphi(3)` will represent respectively loopϕ and closeϕ nodes:

`lphi(loopIndex, initValue, nextValue):` represents the flow of values in a well structured loop with virtual counter `loopIndex`, initial value `initValue` and iteration value `nextValue`.

`cphi(loopIndex, exitExpr, loopVariable):` represents the value of some variable `loopVariable` after crossing a loop identified by `loopIndex` that has an exit condition `exitExpr`.

An Interpreter for SSA Expressions. An interpreter for expressions recorded in `sig(2)` can be implemented using the denotational semantics of the SSA that we have seen in the previous chapter: Figure 3.1 provides a

[1]Remember that we used σ as a container for the declarations in an SSA program: to an SSA identifier σ associates an SSA expression.

PROLOG predicate `int(ssaExpression, iteration, result)` that implements a small interpreter for the SSA. It associates an integer `result` to an `ssaExpression`, for a given `iteration` vector. The first rule in Figure 3.1 is satisfied if the given SSA expression is an integer. In this case the result is the same integer. Otherwise, the second fact is tried. This second fact succeeds only if the given SSA expression is the name of a variable that occurs in `sig(2)`, and then the result is given by interpreting the expression of the declaration. Third and fourth facts are performing arithmetic on integers, fifth fact is satisfied only when the iteration vector contains a zero in the varying loop, that is the initial value in that loop, sixth fact computes the value for all the other iterations in the varying loop by decrementing the iteration vector, and finally the last fact defines the exit value using a minimization operation.

```
int(A, _, A) :- integer(A).

int(A, K, R) :- sig(A, E), int(E, K, R), !.

int(A + B, K, R) :- int(A, K, R1), int(B, K, R2), R is R1 + R2, !.

int(A - B, K, R) :- int(A, K, R1), int(B, K, R2), R is R1 - R2, !.

int(lphi(L, A, _), vect(K, L, 0), R) :- int(A, K, R), !.

int(lphi(L, _, B), vect(K, L, V), R) :- N is V - 1, int(B, vect(K, L, N), R), !.

int(cphi(L, E, A), K, R) :- pmin(K, L, 0, N, E), int(A, N, R), !.
```

Figure 3.1: An SSA interpreter in PROLOG.

This SSA interpreter uses some helper predicates that we now define:

`vect(vector, index, value)` is used for both constructing vectors and for accessing the `value` at `index` in `vector`,

`pmin(vector, index, newVector, expression)` computes a vector with the smallest value for `index` that makes the `expression` become false, informally, this vector represents the iteration exiting the loop, and corresponds to the minimization expression $k[\min\{x \mid \neg\mathcal{E}[\![E_1]\!]\sigma k[x/\ell]\}/\ell]$ defined in the semantics of the SSA presented in the previous chapter. A possible implementation in PROLOG is as follows:

```
pmin(K, L, V, vect(K, L, V), E) :- int(E, vect(K, L, V), 0).
pmin(K, L, V, R, E) :- pmin(K, L, N, R, E), N is V + 1.
```

Note that this implementation of the `pmin(5)` predicate would produce an infinite loop if the expression `E` never evaluates to zero. This matches the infinite behavior of the program containing such a closeϕ expression.

We presented this basic interpreter for illustrating the semantics of the PROLOG predicates that we will use in the remaining of this presentation. We avoided the informal semantics by describing the semantics of the SSA in the PROLOG language, making this operational semantics of the SSA an indirect semantics. The following paragraphs illustrate this semantics with a simple running example.

Distinct Data Circuits. Figure 3.2 provides an SSA program mixing integer values in two cycles: {`a1, a2, a3, a4, a5, a6`} and {`b1, b2, b3`}. The values contained in one of these cycles do not enter in the computation of the values of the other cycle, and thus, these two cycles compute distinct scalar value evolutions, or scalar value streams. This example shows an important aspect of the SSA: scalar variables not belonging to some data stream are completely isolated from the rest of the data streams definitions, and their values can independently be determined.

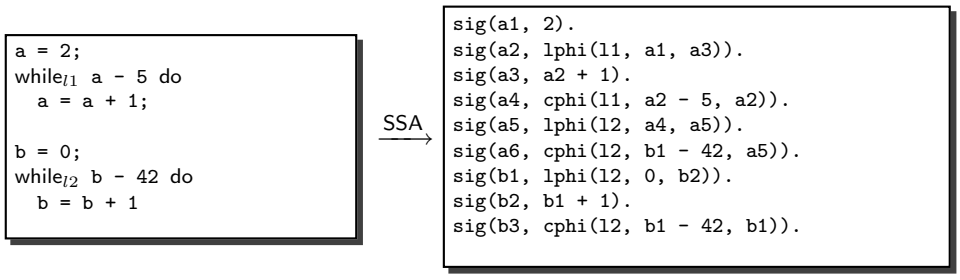


Figure 3.2: An SSA program in PROLOG.

An Illustration of the SSA Operational Semantics. For illustrating the operational mode of the SSA, we will follow some steps of the interpreter on the running example. Suppose that we want to know the value for variable `a4`, then the query to the interpreter predicate would be `int(a4, 0, R)`, in other words, the query is "what is the value `R` of variable `a4` at iteration `0`". PROLOG tries to apply the first rule in the definition of `int(3)`, see Figure 3.1, and fails as `a4` is not an integer. Then the second rule is tried, and the first part of the rule succeeds with the following substitutions:

```
int(a4, 0, R) :- sig(a4, cphi(l1, a2 - 5, a2)), int(cphi(l1, a2 - 5, a2), 0, R).
```

at this point, the second part of the rule `int(cphi(l1, a2 - 5, a2), 0, R)` has to be evaluated to true for the first rule to succeed. This triggers another instance of `int(3)` resolution, as follows: the first rule does not match `int(cphi(l1, a2 - 5, a2), 0, R)`, as there is no relation in `sig(2)` starting with a symbol `cphi(3)`. Next rules are successively tried, and fail as `cphi(3)` is not of the form "`a+b`", or "`lphi(3)`". The last rule partially succeeds with:

```
int(cphi(l1, a2 - 5, a2), 0, R) :- pmin(0, l1, 0, L, a2 - 5), int(a2, L, R).
```

The unification continues with the proof of the right hand side of this equation: the search for a substitution that satisfies `pmin(0, l1, 0, L, a2 - 5)`. After more unification steps, for which we will not give the details, the unification yields the substitution "L = 15" that is applied to the second predicate to be satisfied: `int(a2, L, R)`. The predicate `int(a2, 15, R)` is satisfied with the substitution "R = 17", that satisfies `int(cphi(l1, a2 - 5, a2), 0, 17)`, and that in turn provides the substitution "R = 17" that ends the proof: `int(a4, 0, 17)`.

Once an SSA program is represented in PROLOG, it is possible to use PROLOG for defining symbolic transformations on these predicates. One of the motivations for defining such symbolic transformations is the definition of some optimizing transformation, such as constant propagation, or partial redundancy elimination. In the following we will not discuss these classical optimizing techniques, but we will describe a pattern matching technique that we proposed for eliminating self references from the declarations of SSA variables. For selecting one of the syntactical forms appropriate for our pattern matching algorithm, we define a symbolic transformation that will map SSA syntactic objects with identical meaning to a unique SSA syntactic object.

3.2.3 Condensed SSA Expressions

To define a normal form for SSA expressions, it is possible to define a transformation algorithm that maps different syntactic constructs with equivalent semantics to an identical syntactic object. A PROLOG predicate that implements such a transform contains two arguments: for example `transform(A, B)`, the first argument `A` containing the original expression, and the second argument `B` containing the transformed expression. Given some expression `expr`, it is possible to use this predicate in two different query forms:

- asking for the transformed expression: `transform(expr, B)`,
- or asking for the inverse transform, that is the expression from which this expression has been obtained: `transform(A, expr)`.

Thus, the implementation of the `transform(2)` predicate provides both the transformation algorithm and its inverse, and following the algorithm, a query can generate a family of possible answers, also called a Herbrand universe. The characteristic of a normal form is that the transformation predicate provides a unique idempotent answer in one way, while when employed in the other way, the predicate possibly generates a family of answers: this is illustrated with an example in the following paragraph.

Folding Arithmetic Expressions. In compilers, the role of an arithmetic expression folder is to perform arithmetic computations on statically known scalar operations. A folded arithmetic expression is in its normal form if its size cannot be further reduced. A very primitive arithmetic expression folder is illustrated by the implementation of the `fold(2)` predicate in Figure 3.3.

```
fold(_ + unknown, unknown).
fold(unknown + _, unknown).
fold(L + R, Res) :-
  integer(L), fold(R, ResR), integer(ResR), Res is L + ResR, !;
  integer(R), fold(L, ResL), integer(ResL), Res is ResL + R, !.
fold(L + R, ResL + ResR) :- fold(L, ResL), fold(R, ResR), !.
fold(Default, Default).
```

Figure 3.3: Folding arithmetic expressions.

The first two rules work on undefined elements, and they illustrate the absorbing property of the `unknown` element: used in one way these first two rules of the `fold(2)` predicate define a contraction of the size of expressions, while the other way around, they generate a large family of expressions. The third rule also defines a reduction from left to right, as expressions size is reduced, this time in a statically determined way. The last two rules leave the size of expressions unchanged. Applying any of these rules from left to right guarantees that the expressions size is a monotone decreasing function. This ensures the termination of the computation defined by the `fold(2)` predicate.

Detecting Self Referring Declarations. For defining our non self referring normal form, we need a predicate that detects the self referring declarations in the SSA program. For this, the predicate `hasAself(variable,`

`expression, remainder)` is defined in Figure 3.4, in a way to ease the construction of the normal form transform: the first argument `variable` is the name of the variable for which we try to find an occurrence in `expression`. If an occurrence is found, then `remainder` contains the starting `expression` that was transformed by the `fold(2)` predicate, omitting the first reference to `variable`.

```
hasAself(X, X, 0).
hasAself(X, Name, Step) :-
  sig(Name, Expr), hasAself(X, Expr, Step).
hasAself(X, A + B, Step) :-
  (hasAself(X, A, StepA), fold(StepA + B, Step), !);
  (hasAself(X, B, StepB), fold(A + StepB, Step), !).
```

Figure 3.4: Finding self references.

Normal Form for Self Declarations. Using the previous predicates, we can now define the normalization predicate: Figure 3.5 defines an algorithm `trNorm(ssaDecl, normalForm)` that transforms a declaration of the SSA program `ssaDecl` to its normal form declaration `normalForm`, using the `hasAself(3)` predicate from Figure 3.4 for determining whether the `BackEdge` has a self reference to `X`. If a self reference is found, `BackEdge` is replaced by an equivalent expression `X + Step` where `X` appears in front of the expression, such that it can be simpler matched. The expression obtained in `Step` is the result of the `fold(2)` functor defined in Figure 3.3 that reduces the size of arithmetic expressions by performing basic arithmetic operations on scalar constants.

```
trNorm(sig(X, lphi(LoopId, Init, BackEdge)),
       sig(X, lphi(LoopId, Init, X + Step))) :-
  hasAself(X, BackEdge, Step), !.
trNorm(Default, Default).
```

Figure 3.5: A normalization equivalence.

Illustration. It is possible to apply this normalization transformation to all the declarations of the SSA program. The result of the normalization on the running example, Figure 3.2, is illustrated in Figure 3.6.

It still is possible that `Step` contains self definitions to `X` as the third rule of `hasAself(3)` does not check for self definitions in `B`, if a self reference has

```
sig(a1, 2).
sig(a2, lphi(l1, 2, a2 + 1)).
sig(a3, a2 + 1).
sig(a4, cphi(l1, a2 - 5, a2)).
sig(a5, lphi(l2, a4, a5)).
sig(a6, cphi(l2, b1 - 42, a5)).
sig(b1, lphi(l2, 0, b1 + 1)).
sig(b2, b1 + 1).
sig(b3, cphi(l2, b1 - 42, b1)).
```

Figure 3.6: Normalized SSA for the running example.

been found in `A`: it directly builds the result `StepA + B`. Thus, the expression obtained in `Step` can be quite difficult to handle in general. For avoiding these difficult SSA declarations, we define an algorithm that filters out all these self referring expressions that could produce exponential behaviors.

3.2.4 Abstract SSA in PROLOG

The principal idea behind the use of abstractions when working with the SSA comes from practice: whenever the expression is too complex or not computable at compile time, the use of approximations can help reducing the compiler time, or provide a safe over approximation of the solution. We will illustrate the main ideas by using two classical abstractions: the masking abstractions and the use of abstract elements.

Masking Abstractions. Masks or filters are the most intuitive forms of approximations: a part of the starting representation is destroyed, mapped to an "unknown" element. The masking abstraction is based on a discrete order, that does not relate elements of the representation to each other. For this reason, when a part of the information is replaced by a safe "unknown" approximation, the remaining parts of the representation are still exact. In practice, the techniques of information masking can provide gradual detail levels by defining mask functions that progressively abstract away constructs that are irrelevant for a computation.

```
removeMixers(sig(X, lphi(_, _, X + Step)), sig(X, unknown)) :- hasAself(X, Step, _).
removeMixers(Default, Default).
```

Figure 3.7: Definition of a mask mapping exponentials to an "unknown" element.

Figure 3.7 illustrates the definition of a mask on the declarations of an SSA program: it defines a map of exponential evolutions, for example `sig(x, lphi(l1, 1, x + x))`, matched by the first rule, to an "unknown" element. The last rule identically translates all the remaining constructs not matched by the first rule.

Replacing Symbols with Abstract Elements. The other classical abstraction that we illustrate consists in replacing symbols with a safe description of the values that they represent during the execution of the program. We illustrate this with the translation of the representation of the meet over all paths in terms of an abstract value in Figure 3.8.

```
meetOverAllPaths(sig(X, lphi(_, A, B)), sig(X, meet(A, B))).
meetOverAllPaths(Default, Default).
```

Figure 3.8: Definition of a meet over all paths abstraction.

The first rule replaces the `lphi(3)` predicate in the target representation with the `meet(2)` predicate. The information contained in the loop index disappears, and is replaced by a conservative information that symbolically merges the information contained in both branches of the loopϕ expression.

The `meet(2)` predicate can be defined using classical abstractions, such as for example integer intervals, or sets of values, the choice of the right abstraction being guided by the needs of the computation that uses the abstraction. The number of iterations in the loop containing the loopϕ expression can safely be approximated and the `meet(2)` operation can contain this approximation, in which case it is possible to define a slightly more precise `meet(3)` predicate that takes the approximation of the number of iterations as an extra argument. In the case of self referring variable declarations, the meet operation has to merge symbolic expressions, accounting the number of iterations into the abstraction, whereas in the case of loops with a single iteration, representing condition expressions, the loopϕ expression does not contain self referring symbols, and the meet is a simple operation on values.

We next define a translation technique that replaces self referring loopϕ nodes with an equivalent form, either under the form of lambda functions, or other representations used for polynomial functions.

3.2.5 Characterizing Functions for Scalar Variables

Another possible way to eliminate self references in a precise way, is to translate `lphi(3)` predicates to another representation, called a closed form, that characterizes the sequence of values taken during the execution of the loop. We will define two possible representations for polynomial functions: the lambda function representation, and the chains of recurrences. These representations can then be practically used for data dependence testing, or more generally in restructuring the information in a form that can more systematically be processed. In a next chapter, we will illustrate the use of the closed form descriptions in several applications.

Enriching the Information and Masking Abstractions. Using a different representation than the SSA allows us to modify the semantics of the elements of the SSA that match some extra restrictions, as for example the assumptions of statically decidable non overflowing sequences. The resulting information is more rich than the starting representation, as it can contain the result of a static computation, as for example the determination of the number of iterations in a loop, or the value taken by a variable after crossing a loop. As only a reduced part of the information can be statically translated to the new intermediate form, this translation contains a masking abstraction. The masking abstraction cannot be avoided in this translation to the representation containing a richer information, as the properties needed to a complete translation can be statically undecidable.

We now will analyze a practical example of translation from the SSA expressions to a polynomial form. The resulting representation removes self declarations of loopϕ nodes, and enriches the representation with the result of the static computation of the number of iterations.

Illustration. We use again the running example from Figure 3.2 for illustration. Each one of its scalar variables is associated with a function from loop iteration vectors (i_1, i_2) to integers; as shown in Figure 3.9, these functions can be represented as closed polynomial forms.

Figure 3.10 provides a more complex illustration: variables `b` and `c` are *univariate*, they only depend on one loop counter (i_3), whereas `d` and `e` are *multivariate*. The purpose of induction variable recognition [GSW95] is to statically compute such closed form representations. To compute the evolution of `f`, one must know the *trip count* of the outer loop, i.e., the exact exit value of i_3. To statically evaluate this value, one must already understand the evolutions of `b` and `c`. This example might seem quite artificial: this is due to the fact that the only exit condition that we have chosen to represent

```
sig(a1, 2).
sig(a2, lphi(l1, a1, a3)).
sig(a3, a2 + 1).
sig(a4, cphi(l1, a2 - 5, a2)).
sig(a5, lphi(l2, a4, a5)).
sig(a6, cphi(l2, b1 - 42, a5)).
sig(b1, lphi(l2, 0, b2)).
sig(b2, b1 + 1).
sig(b3, cphi(l2, b1 - 42, b1)).
```

$$
\begin{array}{lcl}
\texttt{a1} & = & 2 \\
\texttt{a2} & = & 2, 3, 4, \ldots \; = \; i_1 + 2 \\
\texttt{a3} & = & 3, 4, 5, \ldots \; = \; i_1 + 3 \\
\texttt{a4} & = & 5 \\
\texttt{a5} & = & 5 \\
\texttt{a6} & = & 5 \\
\texttt{b1} & = & 0, 1, 2, \ldots \; = \; i_2 \\
\texttt{b2} & = & 1, 2, 3, \ldots \; = \; i_2 + 1 \\
\texttt{b3} & = & 42
\end{array}
$$

Figure 3.9: Sequences and closed forms for the running example.

is the test against zero. If the outer loop `l3` in Figure 3.10 runs one more iteration, i.e., the closing phi node is `cphi(l3, b - 4, b)`, the inner loop would not exit in a simple way, as it is exiting in the current example: `d` would be initialized to 4 and would grow by steps of 4, that does not divide 42, and thus, the test against zero of `d - 42` would not be verified. Following the semantics of wrapping types, the inner loop might end, or loop indefinitely: we will discuss this kind of behaviors in Section 4.2, where we will consider a precise semantics of typed operations for machine integers.

```
sig(a, 2).
sig(b, lphi(l3, a, c)).
sig(c, b + 1).
sig(d, lphi(l4, b, e)).
sig(e, d + b).
sig(f, cphi(l3, b - 3, b)).
sig(g, cphi(l4, d - 42, d)).
```

$$
\begin{array}{lcl}
\texttt{a} & = & 2 \\
\texttt{b} & = & 2, 3 \; = \; i_3 + 2 \\
\texttt{c} & = & 3, 4 \; = \; i_3 + 3 \\
\texttt{d} & = & 2, 4, 6, \ldots, 42, 3, 6, 9, \ldots, 42 \; = \; (i_3 + 2)(i_4 + 1) \\
\texttt{e} & = & 4, 6, 8, \ldots, 42, 6, 9, 12, \ldots, 42 \; = \; (i_3 + 2)(i_4 + 2) \\
\texttt{f} & = & 3 \\
\texttt{g} & = & 42
\end{array}
$$

Figure 3.10: Multivariate sequences and closed forms.

As we have seen in this example, the SSA declarations have partitioned the problem such that the resolution only depends on the variables used to define the exit condition, independently of the existence of the inner loop, or any other constructs. The resulting representation under the form of polynomials contains more information than the original SSA form, as for determining this form we have had to prove the termination of the inner loop `l4`, and the non wrapping behavior of the values in the polynomial sequences for every scalar variable.

We now define a more formal representation for polynomial sequences, and the algorithms that construct these representations starting from a program written in SSA form.

Lambda Functions Representation. The first representation of polynomial functions that we will use is based on the lambda expressions, and is represented in PROLOG with the `lambda(2)` predicate: the first argument is the name of the abstraction variable, and the second argument is the expression in which the variable occurs. The translation of a part of the SSA to this representation is implemented by the `fromSSAtoLambda(2)` predicate in Figure 3.11, the first argument is the SSA declaration to be translated, and the second argument is the SSA declaration in which the SSA expression has been replaced by a `lambda(2)` expression.

```
fromSSAtoLambda(sig(X, lphi(_, _, X + Step)), sig(X, unknown)) :- hasAself(X, Step, _).
fromSSAtoLambda(sig(X, lphi(LoopId, Init, X + Step)), sig(X, Init + Result)) :-
  sumNFirst(LoopId, LoopId, Step, Result).

% compute the symbolic sum of the first N iterations of loop Lid: from 0 to N - 1
sumNFirst(_, N, Cst, Cst*lambda(N, binom(N, 1))) :- integer(Cst).
sumNFirst(Lid, N, Cst*lambda(Lid, binom(Lid, K)), Cst*lambda(N, binom(N, ResK))) :-
  integer(Cst), fold(K + 1, ResK).
sumNFirst(Lid, N, A + B, ResA + ResB) :-
  sumNFirst(Lid, N, A, ResA), sumNFirst(Lid, N, B, ResB).
sumNFirst(Lid, N, lambda(Lid, A + B), lambda(Lid, ResA + ResB)) :-
  sumNFirst(Lid, N, A, ResA), sumNFirst(Lid, N, B, ResB).
```

Figure 3.11: From a subset of the SSA language to lambda expressions.

The definition of the `fromSSAtoLambda(2)` predicate uses a helper predicate `sumNFirst(4)` that computes the symbolic sum of the values of the first iterations of a loop by using binomial coefficients in rewriting rules. For detailing the behavior of the `sumNFirst(4)` predicate, we have to consider the following two cases:

- when the `Step` is a constant in the SSA expression that has to be translated: `sig(X, lphi(l, Init, X + Step))`, the first rule of the predicate `sumNFirst(4)` is triggered, yielding:

$$\begin{aligned} X(i) &= \texttt{Init} + \sum_{j=0}^{i-1} \texttt{Step}(j) \\ &= \texttt{Init} + \binom{i}{1} \times \texttt{Step} \end{aligned}$$

- when `Step` is a polynomial of degree $n-1$ with respect to the varying loop `l` in the SSA expression `sig(X, lphi(l, Init, X + Step))`,

and having already determined that `Step` is written under the form $\texttt{Step}(j) = c_0 \times \binom{j}{0} + \ldots + c_{n-1} \times \binom{j}{n-1}$, the second rule of the predicate `sumNFirst(4)` applies, and performs the rewriting rule $\sum_{j=0}^{x-1} \binom{j}{k} = \binom{x}{k+1}$. The resulting expression of the translation ends to be a polynomial of degree n in loop `l`, as illustrated by the following computation:

$$
\begin{aligned}
X(i) &= \texttt{Init} + \sum_{j=0}^{i-1} \texttt{Step}(j) \\
&= \texttt{Init} + \sum_{j=0}^{i-1} (c_0 \times \binom{j}{0} + \ldots + c_{n-1} \times \binom{j}{n-1}) \\
&= \texttt{Init} + \sum_{j=0}^{i-1} \sum_{k=0}^{n-1} (c_k \times \binom{j}{k}) \\
&= \texttt{Init} + \sum_{k=0}^{n-1} (c_k \times \sum_{j=0}^{i-1} \binom{j}{k}) \\
&= \texttt{Init} + \sum_{k=0}^{n-1} (c_k \times \binom{i}{k+1}) \\
&= \texttt{Init} + c_0 \times \binom{i}{1} + \ldots + c_{n-1} \times \binom{i}{n}
\end{aligned}
$$

Figure 3.12 illustrates the translation algorithm to lambda expressions: in the first query the updating expression of the SSA is a simple self referring expression with a step constant in the varying loop; the second query illustrates the effect of applying the `fromSSAtoLambda(2)` predicate on an updating expression that has a step varying in the loop `l1`, and the resulting lambda expression is a polynomial function of degree 2.

```
% | ?- fromSSAtoLambda(sig(a, lphi(l1, 2, a + 2)), Q).
% Q = sig(a, 2+2*lambda(l1,binom(l1,1)))
% | ?- fromSSAtoLambda(sig(a, lphi(l1, 3, a + (4 + 5*lambda(l1,binom(l1,1))))), Q).
% Q = sig(a, 3+(4*lambda(l1,binom(l1,1))+5*lambda(l1,binom(l1,2))))
```

Figure 3.12: Example of translation of SSA to `lambda(2)` expressions.

Chains of Recurrences. Optimizing the storing of the above lambda expressions leads to a representation that only stores the coefficients leading the

binomial coefficients: this representation is called the chains of recurrences [BWZ94, KMZ98, Zim01]. The algorithm that translates a subset of the SSA expressions to this form is defined by the predicate `fromSSAtoMCR(2)` in Figure 3.13: the chains of recurrences are obtained by an exact rewriting of a subset of the SSA language. This remark justifies the use of the chains of recurrences instead of the SSA representation [vE01, vEBS+04] when the SSA representation is not available, or is difficult to be constructed.

```
fromSSAtoMCR(sig(X, lphi(_, _, X + Step)), sig(X, unknown)) :- hasAself(X, Step, _).
fromSSAtoMCR(sig(X, lphi(LoopId, Init, X + Step)), sig(X, mcr(LoopId, Init, Step))).
```

Figure 3.13: From a subset of the SSA language to chains of recurrences.

Figure 3.14 uses the exactly same examples as in Figure 3.12 for illustrating the translation to `mcr(3)` expressions.

```
% | ?- fromSSAtoMCR(sig(a, lphi(l1, 2, a + 2)), Q).
% Q = sig(a,mcr(l1,2,2))
% | ?- fromSSAtoMCR(sig(a, lphi(l1, 3, a + mcr(l1, 4, 5))), Q).
% Q = sig(a,mcr(l1,3,mcr(l1,4,5)))
```

Figure 3.14: Example of translation of SSA to `mcr(3)` expressions.

Chains of recurrences are an alternative representation of polynomial expressions. Previous work [vE01, vEBS+04] try to promote this intermediate representation in static analysis as innovative with respect to anterior techniques, but as we have seen in the previous paragraphs, there is no major improvement over classical lambda expressions: at most, it is possible to acknowledge that the chains of recurrences have the advantage of shortening the expressions size by including the binomial coefficients in their semantics, a useful advantage when analyzing programs with paper and pencil. Thus, there is a single element in favor of preferring chains of recurrences in lieu of their equivalent lambda expressions: chains of recurrences are syntactically shorter than their equivalent lambda expressions.

We shall use the chains of recurrences syntax in the next chapter, for describing a more detailed, yet informal, translation algorithm from the SSA to ASSA based on an imperative language. This informal description will shift our focus from formal definitions of the abstract SSA to more practical questions: the implementation of these static analyzers in a real compiler, the effectiveness of the execution time and memory space needed by the static

analysis, and the use of the information extracted by static analysis in code transformations. These questions are discussed in detail in the next chapter.

3.3 Conclusion

In this chapter we have seen a first definition of the Abstract SSA, or ASSA, a representation that can be built on top of the SSA representation by replacing parts of the SSA with safe approximations. As we will see in the next chapter, this representation will be practical for building other static analyzers, and for statically computing properties of the program. We have seen several algorithms that convert scalar variable declarations in SSA programs to the ASSA representation.

I have proposed a practical way to encode SSA programs using PROLOG predicates, and I have provided an interpreter written in PROLOG for defining the operational semantics of the SSA language. Based on this PROLOG representation of the SSA, I have provided several predicates that translate parts of the SSA to abstract elements. The resulting representation is potentially enriched by the result of static computations, translated to a normal form, and some recursive constructs are translated to practical representations defined under the form of polynomial expressions.

In the next chapter, we will put all these techniques in application, by first providing a high level imperative description of the analysis framework implemented in GCC and then an evaluation of the performances of the implemented static analyses.

Chapter 4

Translation to **ASSA** in Practice

This section provides a description of the translation algorithm from the SSA form as represented in a compiler to the abstract SSA form. More precisely, the chains of recurrences representation will be used as a target representation. This low level description of the analysis algorithms is practical to compiler engineers that would like to implement a similar analyzer in their compiler framework[1]. This description follows the high level lines of the analyzer that I have implemented and integrated in GCC[2]. In the last part of this chapter, I will try to describe the role of this analyzer in GCC, and I will spot several difficult parts, related to the scalar type system, that have to be carefully implemented. We will successively see the compiler infrastructure in which this analyzer can be implemented, the translation algorithm, an illustration of the analyzer when running on two simple examples, and an experimental evaluation of the implementation in GCC.

After the very high level descriptions of the induction variable detection algorithm seen in the previous chapter, I have to warn the reader that the implementation of this same algorithm, which has been integrated in GCC, is written in the C language. This has its drawbacks, as illustrated by the size of the resulting implementation, some 3000 lines of C code[3]. Compared to the implementation in PROLOG that we have seen earlier in this presentation, the C version is much more verbose, handling much more corner cases. The C version of the analyzer is also more efficient in terms of execution time than an equivalent program written in PROLOG. Yet, the most important

[1] A similar analyzer has also been implemented in the LLVM compiler [Lat, LA04], based on one of my first descriptions of the algorithm [PCC+04].

[2] The analyzers have been developed in the loop nest optimizer branch, also known as the *lno-branch*, and then integrated in the main version of GCC during the summer of 2004. The first official version containing these developments is GCC 4.0, released April 20, 2005.

[3] Interested readers can find an implementation in GCC: `tree-scalar-evolution.c`.

point is that the C language is more portable, a strategic point for GCC. In the last part of the chapter, I will describe some experiments that show the central role of the analyzer on optimizations in GCC, and that show the effectiveness and language independence of the implementation. This study will also provide an overview of the code transformations and other static analyzers that have been developed on top of the information extracted by the static analysis of the evolutions of scalar variables.

4.1 Loop Transforms on a Low-Level Representation

Previous work demonstrated the interest of enriched low-level representation [ADvF01, LA04]. They build on the normalization and simplicity of three address code, adding data types, *Static Single-Assignment* form (SSA) [CFR+91, Muc97] to ease data-flow analysis and scalar optimizations, and control and data annotations (loop nesting, heap structure, etc.). Starting from version 4.0, GCC [GCC05] uses such a representation called GIMPLE [Nov03, Mer03], a three-address code derived from SIMPLE, an intermediate representation of the McCAT compiler [HDE+93]. Diego Novillo, from RedHat, and I have proposed the GIMPLE representation in 2001 [Nov] for minimizing the effort in the development and the maintenance of new analyses and transformations. GIMPLE is used for building the SSA representation that is then used by the analyzers and the scalar and loop nest optimizers. After these optimizations, the intermediate representation is translated into the Register Transfer Language (RTL), where machine-specific informations are added to the representation.

Structure of GCC. Figure 4.1 presents the overall structure of GCC. The front-ends C, C++, Java, Fortran95 and Ada are translating their intermediate representation to a common representation called GENERIC [Mer03]. GENERIC contains a set of the imperative language constructs needed to efficiently translate any imperative specific construct used by the front-end languages. At the moment of this writing, there is no formal description of the semantics of GENERIC. The translation of each front-end language to GENERIC provides the only precise, indirect definition[4] of the GENERIC

[4]The maps between front-end constructs with known semantics and the GENERIC constructs formally define GENERIC constructs in an indirect way, as the semantics of GENERIC is defined in function of the semantics of the constructs of the front-end languages semantics. Furthermore, the current implementations of the translation of front-

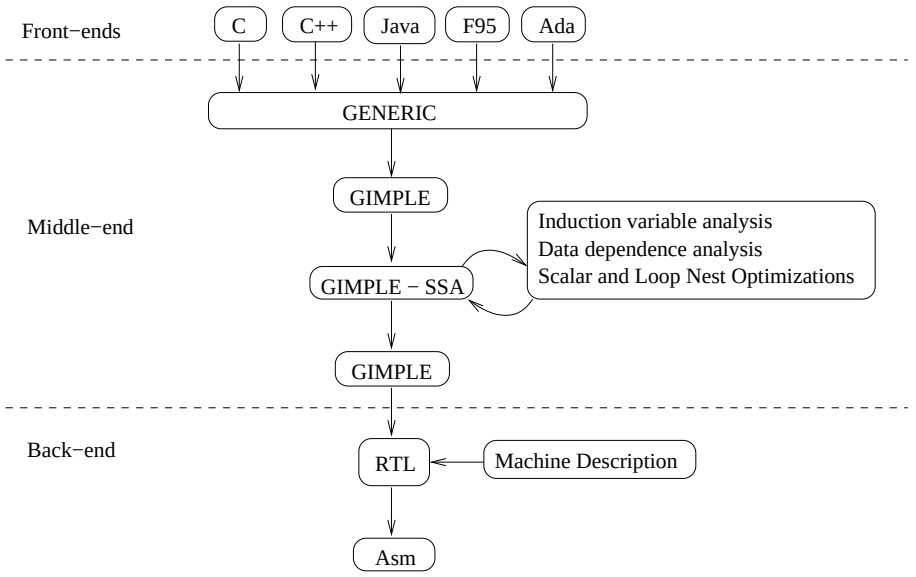


Figure 4.1: The infrastructure of GCC 4.

language. This definition of GENERIC is not the best documentation for the implementation of new language front-ends, but it is the only accurate formal description.

Continuing our description of the flow of information inside GCC, the GENERIC language is then translated to the GIMPLE language by decomposing expressions into a three-address code, and replacing loop constructs with their goto equivalent constructs. The control flow graph, CFG, is built on top of this low level representation, and the SSA form is then built on top of this representation. The natural loops [ASU86] are discovered as strongly connected components on top of the CFG, and these loop structures are used for the analysis of induction variables that we will describe in this section. Natural loops are represented as annotations in the examples of code that we will see through this chapter, but they actually are implemented as a data structure on top of the CFG, containing pointers to the header of the loop, to the back edge, and to the exit edge. A variable i_k will represent the implicit counter[5] associated with the loop number k, *loop* (i_k). Loop annotations are enriched with the results of the static computation of the number of iterations: corresponding to the smallest value of the loop counter

ends abstract syntax trees to GENERIC are written in the C language. Thus, the semantics of the translation is also relative to the semantics of the C language.

[5]There is no correspondence between the variables that we will use for loop counters and the variables declared in the analyzed program.

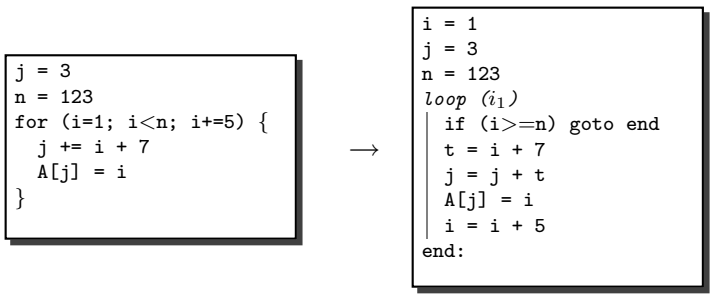


Figure 4.2: Lowering to three-address code.

for which one of the loop exit conditions become true. Practically this could be implemented as the resolution of a constraints system, but for the moment GCC uses a very restrictive algorithm that considers only single exit loops.

After several analyses and code transformations have been performed on this representation, the code is rewritten out of the SSA form, and then GIMPLE representation is eventually translated to RTL that finally is translated to assembler based on the machine instructions patterns.

Specificities of the GIMPLE Representation. In a three-address representation like GIMPLE, subscript expressions, loop bounds and strides are spread across a number of instructions and basic blocks, possibly far away from the original location in the source code. This is due to the lowering itself and to intermediate optimization phases, such as dead-code elimination, partial redundancy elimination, optimization of the control flow, invariant code motion, etc. Figure 4.2 illustrates this increased variability and complexity.

A three-address representation like GIMPLE could seem not suitable to implement program transformations for high performance: classical loop nest transformations were described on FORTRAN DO loop programs as source to source transformations, whereas in GIMPLE, control flow is represented by only two primitives, a conditional expression `if`, and a goto expression `goto`. Loops have to be discovered from the control flow graph [ASU86]. Although the number of iterations is not captured by the GIMPLE syntax, it is often discovered by analyzing the scalar variables involved in the loop exit conditions; this allows to compute precise information lost in the translation to a low-level representation, but it may also be useful when the source level does not expose enough syntactic information, e.g., in `while` loops, or loops with irregular control constructs such as `break` or exceptions. Even if in GIMPLE, subscript expressions, loop bounds and strides are spread across several elementary instructions and basic blocks, the SSA form provides fast

practical ways to extract information concerning values of scalar variables.

Induction Variable Analyses. Lowering the loop to three-address code makes the inductive definitions more intricate as shown on the right-hand side of Figure 4.2. Looking at the original loop in the left-hand side, variable `j` is inductively updated with a non-negative value, meaning it is a strictly increasing function of the loop counter `i`. Indeed, `i` takes the values $1, 6, 11, \ldots, 126$ and `j` is successively evaluated to $3, 11, 24, \ldots, 1703$. This observation is sufficient to prove there are no dependences on array `A`. The evolution of `i` and `j` may also be characterized as closed form expressions, parameterized by the number of iterations i_1 of the enclosing loop.

$$\begin{aligned} \texttt{i} &= 5i_1 + 1, \\ \texttt{j} &= \frac{5}{2}{i_1}^2 + \frac{11}{2}i_1 + 3. \end{aligned}$$

Although the evolution of `j` is not an affine (or linear) expression, several analyses would retrieve enough information to detect the absence of data dependence [Wol92, GSW95, HP96, WCP01, vE01, vEBS+04, RZR04, CT04]. The presentation of all these analyses is informal: the starting language is either a subset of a programming language, such as FORTRAN or C, or is not defined at all. In all the cited papers, the translation between the starting language and the target language uses informal semantics, presenting an informal algorithm for the translation between two informally defined languages, and illustrating the translation process with a running example. The target language receives the most attention, and is regarded as the original part of the work: one of the metrics used for comparing against the previous work is the expressiveness of the target language. However, making the target language more and more expressive is contrary to the goal of static analysis.

The originality of the work that I present in this thesis is based on the more classical point of view on the induction variable analysis: starting from a formal semantics of the source representation, the SSA language, filtering out several difficult elements, or selecting only a part of the SSA constructs, produces a subset of the SSA called TREC, a language enough expressive for capturing frequent programming constructs. A second abstracting process translates the TREC into less expressive representations such as the affine functions. We have seen a formal presentation of these abstracting processes in the previous chapter. The following presentation is intendedly an informal description of the abstracting algorithms presented in the previous chapter. These abstracting processes are performed on the demand of a user pass of

this information, like the data dependence analysis or the vectorizer. Finally, speed, robustness of the implementation and language-independence are natural benefits of using a low-level static single assignment language.

SSA Form. In the reminder of this chapter, we will use a definition of the SSA that is much closer to the informal definition given in the seminal papers on the SSA, than the definition of the SSA that we have seen in the last chapter, see [CFR+91, Muc97] for details: the SSA graph is the graph of use to definition chains in the SSA representation; ϕ nodes (or functions) occur at merge points and restore the flow of values from the renamed variables; the ϕ arguments are variables listed in the order of the associated control flow edges; ϕ nodes are split into the loopϕ — nodes whose second argument correspond to a back-edge in the control flow graph — and condϕ categories. We sometimes will note loop$_1\phi$ for a loopϕ node defined in loop 1.

As we have seen in Chapter 2, in the SSA representation a part of the statically available information contained in the Dewey-like numbers was transformed. Although the loop nesting information should be intrinsic to the SSA, in the classical SSA form it is not easy to reconstruct this information only from the SSA representation, and thus for practical reasons, we will use the loop nesting information in conjunction with the SSA representation for presenting the algorithms working on SSA. In particular, in order to identify in the classical SSA representation the loop that contains some loopϕ node, we use the loop hierarchy.

Introductory Examples. To illustrate the main issues and concepts, we now will consider several examples. The closed-form expression for `f` in Figure 4.3 is not affine (a second degree polynomial).

```
a = 3
b = 1
loop (i1)
| c = loop1φ (a, f)
| d = loop1φ (b, g)
| if (d>=123) goto end
| e = d + 7
| f = e + c
| g = d + 5
end:
```

At each step of the loop, an integer value following the sequence $1, 6, 11, \ldots, 126$ is assigned to d, that is the affine function $5i_1 + 1$; a value in the sequence $3, 11, 24, \ldots, 1703$ is assigned to f, that is a polynomial of second degree: $\frac{5}{2}{i_1}^2 + \frac{11}{2}i_1 + 3$.

Figure 4.3: First example: polynomial functions.

In Figure 4.4, apart from `d`, all variables are *univariate*: they only depend on one loop counter; `d` is called *multivariate*. To compute the evolution of `c`, `x` and `d` in the second example, one must know the *trip count* of the inner

```
a = 3
loop (i1)
| c = loop1φ (a, x)
| loop (i2)
| | d = loop2φ (c, e)
| | e = d + 1
| | t = d - c
| | if (t>=9) goto end2
| end2:
| x = e + 3
| if (x>=123) goto end1
end1:
```

The successive values of c are $3, 17, 31, \ldots, 115$, that is the affine univariate function $14i_1 + 3$. The successive values of x in the loop are $17, 31, \ldots, 129$ that is $14i_1 + 17$. The evolution of variable d, $3, 4, 5, \ldots, 13, 17, 18, 19, \ldots, 129$ depends on the iteration number of both loops: that is the multivariate affine function $14i_1 + i_2 + 3$.

Figure 4.4: Second example: multivariate functions.

loop, here ten iterations. Yet, to statically evaluate the trip count of i_2 one must already understand the evolutions of `c` and `d`.

```
loop (i1)
| a = loop1φ (1, b)
| if (a>=100) goto end1
| b = a + 4
| loop (i2)
| | c = loop2φ (a, e)
| | e = loop2φ (b, f)
| | if (e>=100) goto end2
| | f = e + 6
| end2:
end1:
```

The sequence of values taken by a is $1, 5, 9, \ldots, 101$ that can be written in a condensed form as $4i_1 + 1$. The values taken by variable e are $5, 11, 17, \ldots, 95, 101, 9, 15, 21, \ldots, 95, 101$ and generated by the multivariate function $6i_2 + 4i_1 + 5$. These two variables are used to define the variable c, that will contain the successive values $1, 5, 11, \ldots, 89, 95, 5, 9, 15, \ldots, 89, 95$: the first value of c in the loop i_2 is the value coming from a, while the subsequent values are those of variable e.

Figure 4.5: Third example: wrap-around.

In the third example, Figure 4.5, variables `a` and `e` have simple affine evolutions, but `c` is a typical case of *wrap-around* variable [Wol92], a variable that takes a special value at the first iteration and becomes an inductive variable with a regular evolution in further iterations. Such variables are defined by loopϕ nodes appearing in strongly-connected components of the SSA graph that hold *other* loopϕ nodes. The classification by Wolfe et al. [GSW95] does not handle wrap-around variables in nested loops where the initial value is varying in the outer loops. Indeed, the closed form representation for such *multivariate* variables are more complex than the affine, polynomial or exponential cases considered in [Wol92, GSW95]. Here the value of `c` is reinitialized to a different value at each iteration of the outer loop: in the first iteration of i_1, its values in the inner loop are $1, 5, 11, 17$, then in the second iteration of i_1, its values are $5, 9, 15, 21, \ldots$, etc. We will see

the closed form of c in Section 4.2, after the introduction of a representation that captures such evolutions.

Some complex SSA graphs with multiple loopϕ nodes in a cycle do not match the wrap-around class. They define more general induction variables that include the *periodic* or *exponential* classes [Wol92], or may not even have a known closed form.[6]

```
loop (i1)
  a = loop1φ (0, d)
  b = loop1φ (0, c)
  if (a>=100) goto end
  c = a + 1
  d = b + 1
end:
```

Both a and b have affine evolutions: $0, 1, 2, \ldots, 100$, because they both have the same initial value. However, if their initial value is different, their evolution can only be described by a periodic evolution.

Figure 4.6: Fourth example : periodic evolution functions.

Figure 4.6 presents an example where a and b have a linear closed form but derive from an intricate inductive definition scheme.

```
loop (i1)
  a = (unsigned char) loop1φ (0, c)
  b = (int) loop1φ (0, d)
  c = (unsigned char) (a + 1)
  d = (int) (b + 1)
  if (d >= 1000) goto end
  T[b] = U[a]
end:
```

The C programming language defines modulo arithmetics for unsigned typed variables. In this example, the successive values of variable a are periodic: $0, 1, 2, \ldots, 255, 0, 1, \ldots$, or in a condensed notation $i_1 \pmod{256}$.

Figure 4.7: Fifth example: effects of types on the evolution of scalar variables.

```
loop (i1)
  a = (char) loop1φ (0, c)
  b = (int) loop1φ (0, d)
  c = (char) (a + 1)
  d = (int) (b + 1)
  if (d > N) goto end
end:
```

Signed types overflow are not defined in C. The behavior is only defined for the values of a in $0, 1, 2, \ldots, 126$, consequently d is only defined for $1, 2, 3, \ldots, 127$, and the loop is defined only for the first 127 iterations.

Figure 4.8: Sixth example: inferring properties from undefined behavior.

Figure 4.7 illustrates an unusual data dependence problem. Variable a is incremented at each iteration of i_1, however the *unsigned char* type constraints its evolution to the range $[0, 255]$. When language standards define

[6] A complete algebra does not exist for the Turing-complete computation model of scalar evolutions.

modulo arithmetics for a type, the compiler has to handle the effects of wrapping overflows on induction variables. When the effect of overflowing operations is not defined by the language as wrapping, then based on the defined part of the domain, the compiler is allowed to deduce constraints on the values of scalar variables, or to infer safe bounds for loops, as illustrated in Figure 4.8.

4.2 Trees of Recurrences

In this section, we introduce the notion of *Tree of Recurrences* (TREC), a closed-form that captures the evolution of induction variables as a function of iteration indices and allows an efficient computation of values at given iteration points. This formalism extends the expressive power of *Multivariate Chains of Recurrences* (MCR) [BWZ94, KMZ98, Zim01, vE01] by symbolic references. MCR are obtained by an abstraction operation that instantiate all the varying symbols: some evolutions are mapped to a "don't know" symbol $\top$.

Let $F(i_1, i_2, \ldots, i_m)$, or $F(\vec{\ell})$, represent the evolution of a variable inside a loop of depth m as a function of $i_1, i_2, \ldots, i_m$. F can be written as a closed form Θ, called TREC, that can be statically processed by further analyses and efficiently evaluated at compile-time. The syntax of a TREC is derived from MCR and inductively defined as: $\Theta = \{\Theta_a, +, \Theta_b\}_k$ or $\Theta = c$, where Θ_a and Θ_b are trees of recurrences and c is a constant or a variable name, and subscript k indexes the dimension. As a form of syntactic sugar, $\{\Theta_a, +, \{\Theta_b, +, \Theta_c\}_k\}_k = \{\Theta_a, +, \Theta_b, +, \Theta_c\}_k$.

Evaluation of TREC. The value $\Theta(i_1, i_2, \ldots, i_m)$ of a TREC Θ is defined as follows: if Θ is a constant c then $\Theta(\vec{\ell}) = c$, else Θ is $\{\Theta_a, +, \Theta_b\}_k$ and

$$\Theta(\vec{\ell}) = \Theta_a(\vec{\ell}) + \sum_{l=0}^{i_k-1} \Theta_b(i_1, \ldots, i_{k-1}, l, i_{k+1}, \ldots, i_m) \ .$$

The evaluation of $\{\Theta_a, +, \Theta_b\}_k$ for a given $\vec{\ell}$ matches the inductive updates across i_k iterations of loop k: Θ_a is the initial value, and Θ_b the increment in loop k. This is an exponential algorithm to evaluate a TREC, but [BWZ94] gives a linear time and space algorithm based on Newton interpolation series. Given a univariate MCR with c_0, c_1, ..., c_n, constant parameters (either scalar constants, or symbolic names defined outside loop k):

$$\{c_0, +, c_1, +, c_2, +, \ldots, +, c_n\}_k(\vec{\ell}) = \sum_{p=0}^{n} c_p \binom{\ell_k}{p} \ . \tag{4.1}$$

This result comes from the following observation: a sum of multiples of binomial coefficients — called *Newton series* — can represent any polynomial. The closed form for `f` in the first example of Figure 4.3 is the second order polynomial $F(i_1) = \frac{5}{2}{i_1}^2 + \frac{11}{2}i_1 + 3 = 3\binom{i_1}{0} + 8\binom{i_1}{1} + 5\binom{i_1}{2}$, and is written $\{3, +, 8, +, 5\}_1$. The coefficients of a TREC derive from a finite differentiation table: for example, the coefficients for the TREC associated with $\frac{5}{2}{i_1}^2 + \frac{11}{2}i_1 + 3$ can be computed either by differencing the successive values [HP96]:

i_1	0	1	2	3	4
c_0	**3**	11	24	42	65
c_1	**8**	13	18	23	
c_2	**5**	5	5		
c_3	**0**	0			

or, by directly extracting the coefficients from the code [vE01]. Section 4.4 will present the algorithm for extracting TREC from an SSA representation.

The fast evaluation of a TREC is illustrated on the second introductory example, in Figure 4.4, where the evolution of `d` is the affine equation $F(i_1, i_2) = 14i_1 + i_2 + 3$. A TREC for `d` is $\Theta(i_1, i_2) = \{\{3, +, 14\}_1, +, 1\}_2$, that can be evaluated for $i_1 = 10$ and $i_2 = 15$ as follows:

$$\Theta(10, 15) = \{\{3, +, 14\}_1, +, 1\}_2(10, 15) = 3 + 14 \cdot \binom{10}{1} + \binom{15}{1} = 158 \ .$$

Arithmetic Operations. Arithmetic operations on MCR are defined as rewriting rules [vE01]. They trivially derive from the arithmetic operations on polynomials. A reduced set of these rewriting rules are defined in Figure 4.9.

$$\begin{aligned}
x + \{\Theta_a, +, \Theta_b\}_k &= \{x + \Theta_a, +, \Theta_b\}_k \\
x \times \{\Theta_a, +, \Theta_b\}_k &= \{x \times \Theta_a, +, x \times \Theta_b\}_k \\
\{\Theta_a, +, \Theta_b\}_k + \{\Theta_c, +, \Theta_d\}_k &= \{\Theta_a + \Theta_c, +, \Theta_b + \Theta_d\}_k \\
\{\Theta_a, +, \Theta_b\}_k \times \{\Theta_c, +, \Theta_d\}_k &= \{\Theta_a \times \Theta_c, +, \{\Theta_a, +, \Theta_b\}_k \times \Theta_d \\
&\qquad + \{\Theta_c, +, \Theta_d\}_k \times \Theta_b + \Theta_b \times \Theta_d\}_k
\end{aligned}$$

Figure 4.9: Arithmetic operations on TREC: Θ_a, Θ_b, Θ_c and Θ_d are trees of recurrences, and x is a constant.

Instantiation of TREC and Abstract Envelopes. In order to be able to use the efficient evaluation scheme presented above, symbolic coefficients of a TREC have to be analyzed: the role of the instantiation pass is to limit the expressive power of TREC to MCR. Difficult TREC constructs such as exponential self referring evolutions are either translated to some appropriate representation, or discarded[7].

Because a large class of optimizers and analyzers are expecting simpler cases, TREC information is filtered using an instantiation pass. Several abstract views can be defined by different instantiation passes, such as mapping every non polynomial scalar evolution to $\top$, or even more practically, mapping non affine functions to $\top$. In appropriate cases, it is natural to map uncertain values to an abstract value: we have experimented instantiations of TREC with intervals, in which case we obtain a set of possible evolutions that we call an envelope. Allowing the coefficients of TREC to contain abstract scalar values is a more natural extension than the use of maximum and minimum functions over MCR as proposed by van Engelen in [vEBS+04] because it is then possible to define other kinds of envelopes using classical scalar abstract domains, such as polyhedra, octagons [Min01], or congruences [Gra91].

Peeled trees of recurrences. A frequent occurring pattern consists in variables that are initialized to a value during the first iteration of a loop, and then is replaced by the values of an induction variable for the rest of iterations. We have chosen to represent these variables by explicitly listing the first value that they contain, and then the evolution function that they follow. The *peeled* TREC are described by the syntax $(a, b)_k$ whose semantics is given by:

$$(a, b)_k(x) = \begin{cases} \text{a} & \text{if } x = 0, \\ \text{b (x - 1)} & \text{for } x \geq 1, \end{cases}$$

where a is a TREC with no evolution in loop k, b is a TREC that can have an evolution in loop k, and x is indexing the iterations in loop k. Most closed forms for wrap-around variables [Wol92] are peeled TREC. Indeed, back to the introductory example in Figure 4.5, the closed form for c can be represented by a peeled multivariate affine TREC: $(\{1, +, 4\}_1, \{\{5, +, 4\}_1, +, 6\}_2)_2$.

A peeled TREC describes the first values of a closed form chain of recurrence. In some cases it is interesting to replace it by a simpler MCR, and vice versa, to peel some iterations out of a MCR. For example, the peeled TREC $(0, \{1, +, 1\}_1)_1$ describes the same function as $\{0, +, 1\}_1$. This last form is

[7]Optimizers such as symbolic propagation could handle such difficult constructs, however they lead to problems that are difficult to solve in practice.

a unique representative of a class of TREC that can be generated by peeling one or more elements from the beginning. Simplifying a peeled TREC amounts to the unification of its first element with the function represented in the right-hand side of the peeled TREC. A simple unification algorithm tries to add a new column to the differentiation table without changing the last element in that column. Since this first column contains the coefficients of the TREC, the transformation is possible if it does not modify the last coefficient of the column, as illustrated in Figure 4.10.

i_1	0	1	2	3	4
c_0	3	11	24	42	65
c_1	8	13	18	23	
c_2	5	5	5		
c_3	0	0			

i_1	0	1	2	3	4	5
c_0	**0**	3	11	24	42	65
c_1	**3**	8	13	18	23	
c_2	**5**	5	5	5		
c_3	**0**	0	0			

Figure 4.10: Adding a new column to the differentiation table of the chain of recurrence $\{3, +, 8, +, 5\}_1$ leads to the chain of recurrence $\{0, +, 3, +, 5\}_1$.

This technique allows the unification of 29 wrap around loop-ϕ in the SPEC CPU2000, 337 on the GCC code itself, and 5 on the JavaGrande.

Finally, we formalize the notion of peeled TREC equivalence class: given integers v, $a_1, \ldots, a_n$, a TREC $c = \{a_1, +, \ldots, +, a_n\}_1$, a peeled TREC $p = (v, c)_1$, and a TREC $r = \{b_1, +, \ldots, +, b_{n-1}, +, a_n\}_1$, with the integer coefficients $b_1, \ldots, b_{n-1}$ computed as follows: $b_{n-1} = a_{n-1} - a_n$, $b_{n-2} = a_{n-2} - b_{n-1}$, ..., $b_1 = a_1 - b_2$, we say that r is equivalent to p if and only if $b_1 = v$.

Periodic trees of recurrences. Periodic sequences may be generated by flip-flop operations, that are special cases of self referenced peeled TREC. Variables in a flip-flop exchange their initial values over the iterations, for example:

$$flip \to (3, 5, flip)_k(x) = [3, 5]_k(x) = \begin{cases} 3 & \text{if } x = 0 \pmod 2, \\ 5 & \text{if } x = 1 \pmod 2. \end{cases}$$

Typed trees of recurrences. Most compilers handle ad-hoc type schemes for induction variables. We will see a typed extension of the TREC that is both sound and accurate, by delaying the interpretation of complex type annotations and conversions to the point where the nature of the needed abstraction is known (intervals, affine forms, etc.). First, we will see an intuitive idea of the behavior of typed sequences, then we will see a more formal definition as an algorithm that converts typed sequences to congruence relations.

One of the solutions for preserving the semantics of wrapping types on TREC operations is to type the TREC and to map the effects of types from the SSA representation to the TREC representation. For example, the conversion from *unsigned char* to *unsigned int* of TREC $\{(uchar)100, +, (uchar)240\}_1$ is $\{(uint)100, +, (uint)\texttt{0xffffffff0}\}_1$, such that the original sequence remains unchanged $(100, 84, 68, \ldots)$. The first step of a TREC conversion proves that the sequence does not wrap. In the previous example, if the number of iterations in loop 1 is greater than 6, the converted TREC should also contain a wrap modulo 256, as illustrated by the first values of the sequence: $100, 84, 68, 52, 36, 20, 4, 244, 228, \ldots$.

When it is impossible to prove that a sequence does not wrap, it is safe to assume that it wraps, and it is possible to propose several solutions for representing sequences that potentially wrap: the exact description of the sequence, and several abstractions.

The first solution is to use a periodic TREC, that lists all the values in a period: in the previous example we would have to store 15 values. Using periodic TREC for sequences wrapping over narrow types can seem practical, but this method is not practical for arbitrary sequences over wider types.

Instead of using an exhaustive description of the sequence, it is possible to use an approximated representation: the interval representation of an overflowing sequence would be the whole range of the type: in the previous example, $[0, 255]$. This precision is not sufficient for proving that the values of the sequence cannot be zero. However, on a more precise representation, such as the congruence relations [Gra91], it is possible to prove that no value of the sequence can be zero: the sequence is represented as the set of integers $x \in [0, 255]$ that verify $x = 4 \pmod{16}$.

The most accurate representation is close to the source representation, and contains the cast operations: $(uint)(\{(uchar)100, +, (uchar)240\}_1)$. Because the right abstract representation can only be determined based on the precision of a computation, the computation of these casts are delayed.

More formally, we now will see a denotational semantics for typed TREC as mappings from integer sequences to sequences over $\mathbb{Z}/n\mathbb{Z}$ for some integer n. A language for typed TREC is defined by including a cast operation:

$$
\begin{array}{rcl}
(Sequences) & & \\
Seq & ::= & (Type)Seq \mid \mathsf{TREC} \\
(Types) & & \\
Type & ::= & char \mid uchar
\end{array}
$$

In this presentation, the considered types *char* and *unsigned char* will

have a wrapping semantics, matching the semantics of these types in the Java language. The denotational semantics for this language is:

$$\begin{aligned} \mathcal{S}[\![(uchar)a]\!] &= \lambda x.\mathcal{S}[\![a]\!](x) \pmod{2^8} \\ \mathcal{S}[\![(char)a]\!] &= \lambda x.(\mathcal{S}[\![a]\!](x) + 2^7) \pmod{2^8} - 2^7 \end{aligned}$$

In general, the semantics of a sequence over a wrapping type with a range $[min, max]$ is $\lambda x.((\mathcal{S}[\![a]\!](x) - min) \pmod{max - min}) + min$. Conversion from a type to another is performed using arithmetic modulo the width of the type: $max - min$. In the algorithm presented in Section 4.4 we will use a CONVERT function defined following the semantics of type conversion for the programming language.

In practice, it is not convenient to work with the above definition of the overflowing types, as the precision of the target types might exceed the precision of the available integers on the machine that hosts the compiler. A possible solution is the use of multiple precision library, but the cost of arithmetic in multiple precision mode is not affordable in practice. The implementation of conversion of sequences in GCC is based on an ad-hoc handling of integer types conversions, such that any operation on typed value do not exceed the precision of two double integers.

Exponential Trees of Recurrences. The exponential MCR [BWZ94] used by [vE01] and then extended to handle sums or products of polynomial and exponential evolutions [vEBS+04] are useless in compiler technology for typed integer sequences, as integer typed arithmetic has limited domains of definition. Any overflowing operation either has defined modulo semantics, or is not defined by the language standard. The longer exponential integer sequence that can exist for an integer type of size 2^n is $n-1$: left shifting the first bit $n-2$ times. Storing exponential evolutions as peeled TREC seems efficient, because in general $n \leq 64$.

Next, we will see an efficient algorithm that translates a part of the SSA dealing with scalar variables to TREC.

4.3 Abstract SSA on GIMPLE

I now present an algorithm that implements the scalar evolutions analyzer: this algorithm computes closed-form expressions for some inductive scalar variables. The interface to this analyzer is designed as an interface to a database that contains, for a given variable definition, its evolution function

under the form of a TREC. For example, when the data dependence analyzer needs the evolution function of a variable that indexes an array, it simply queries the database that either returns the cached previously computed evolution function, or otherwise triggers the analysis of the asked variable, triggering the analysis of all the variables, loop counts, etc., needed to determine the evolution function.

Several constraints have led the design of this analyzer:

- first, the algorithm does not assume a particular control flow structure and makes no restriction on the recursive intricate variable definitions. It however fails to detect any meaningful induction variable on irreducible control flow graphs that cannot be analyzed into natural loop structures [ASU86]. For all the variables defined in one of the basic blocks of an irreducible region, the answer of the analyzer will be the value $\top$ that stands for an uncomputable evolution,

- another characteristic of this algorithm is that it does not use the syntactic information of the analyzed SSA representation. In other words, it makes no distinction between the names of variables defined in the source code and those that are introduced by the lowering to three-address code, or by other optimizers,

- furthermore, the algorithm is able to delay a part of the analysis until more information is known, by leaving symbolic names in the representation. The representation that is obtained from AnalyzeEvolution function is the most instantiated with respect to the instantiation context, that is, no early approximations have been performed. Based on this representation, symbolic solvers, as for example the computation of the number of iterations in a loop, may produce safe and precise informations that improve the information available in the instantiation context,

- the last constraint that is important for the inclusion of an implementation of the algorithm in a production compiler is that the analyzer should be linear in time and space. In order to satisfy this constraint and to allow further possible refinements, an interface provides views of different levels of abstractions. This can practically be implemented by several procedures that instantiate TREC.

The structure of the analysis algorithm is quite complex because it is based on a double recursion as sketched in Figure 4.11. It presents similarities with the algorithm for linear unification [PW76], where the double recursion is hidden behind a single recursion with a stack structure.

Figure 4.11: Bird's eye view of the analyzer

4.4 Algorithm

Figure 4.12 presents a driver application ComputeLoopPhiEvolutions, that computes a TREC for every variable whose value is alive across loop iterations. In practice, the computation of closed form expressions are triggered by applications like the dependence analysis or the evaluation of loop-trip count. As illustrated in this driver, the applications call the analyzer AnalyzeEvolution, presented in Figure 4.13, for a given loop and a variable name. The results are then filtered through an abstraction function InstantiateEvolution presented in Figure 4.15.

Algorithm: ComputeLoopPhiEvolutions
Input: SSA representation of the procedure
Output: a TREC for every variable defined by loopϕ nodes
  For each loop l in a depth-first traversal of the loop nests
    For each loopϕ node n in loop l
      InstantiateEvolution(AnalyzeEvolution(l,n), l)

Figure 4.12: Driver application.

The first step of AnalyzeEvolution is a query to the database for the evolution function of the analyzed variable. The database is only visible to the AnalyzeEvolution function and is accessed using the construct, Evolution[n], for an SSA name definition n. The value contained initially in the database for a non analyzed variable name is $\perp$. The database ensures that the analysis is performed only once for a given variable name. The main part of the analyzer consists in a pattern matching of five common expressions occurring in a three-address SSA representation, with the corresponding associated action. The first pattern, "`v = constant`", is the simplest one: the resulting evolution is constant. The second pattern, "`v = a`", propagates the evolution function by copy. If the analyzer is restricted to these two patterns, the analyzer has the same role and expressive power as a constant propagation pass. To these basic patterns is added an interpreter,

Algorithm: AnalyzeEvolution(l, n)
Input: l the current loop, n the definition of an SSA name
Output: TREC for the variable defined by n within l
  v ← variable defined by n
  ln ← loop of n
  If Evolution[n] $\neq \bot$ Then res ← Evolution[n]
  Else If n matches "`v = constant`" Then res ← `constant`
  Else If n matches "`v = a`" Then res ← AnalyzeEvolution(l, `a`)
  Else If n matches "`v = (type) a`" Then
    res ← Convert(`type`, AnalyzeEvolution(l, `a`))
  Else If n matches "`v = a` $\odot$ `b`" (with $\odot \in \{+, -, *\}$) Then
    res ← AnalyzeEvolution(l, `a`) $\odot$ AnalyzeEvolution(l, `b`)
  Else If n matches "`v = loop`ϕ`(a, b)`" Then
    (notice `a` is defined outside loop ln and `b` is defined in ln)
    Search in depth-first order a path from `b` to `v`:
    ($exist$, $update$) ← BuildUpdateExpr(n, definition of `b`)
    If not $exist$ (if such a path does not exist) Then
      res ← $(\mathtt{a}, \mathtt{b})_l$: a peeled TREC
    Else If $update$ is $\top$ Then res ← $\top$
    Else res ← $\{\mathtt{a}, +, update\}_l$: a TREC
  Else If n matches "`v = cond`ϕ`(a, b)`" Then
    eva ← InstantiateEvolution(AnalyzeEvolution(l, `a`), ln)
    evb ← InstantiateEvolution(AnalyzeEvolution(l, `b`), ln)
    res ← Meet(eva, evb)
  Else res ← $\top$
  Evolution[n] ← res
  Return res

Figure 4.13: Main analyzer.

the third pattern "`v = a` $\odot$ `b`", that maps the arithmetic operations of the source language onto the arithmetic operations of the target language. Note that cast operations can be implemented as part of this interpreter, but are not presented in Figure 4.13 for simplifying the presentation. With this extension, the analyzer is slightly more expressive than the classical constant propagation because it is also able to fold some of the arithmetic expressions into constants.

The cornerstone of the analyzer is in the fourth pattern, "`v = loop`ϕ`(a, b)`", that analyses ϕ nodes whose arguments are defined at different loop

Algorithm: BUILDUPDATEEXPR(h, n)
Input: h the halting loopϕ, n the definition of an SSA name
Output: (*exist*, *update*), *exist* is true if h has been reached,
update is the reconstructed expression for the overall effect in loop of h
  If (n is h) Then Return (true, 0)
  Else If n is a statement in an outer loop Then Return (false, $\perp$),
  Else If n matches "`v = a`" Then
    Return BUILDUPDATEEXPR(h, definition of `a`)
  Else If n matches "`v = (type) a`" Then
    (*exist*, *update*) $\leftarrow$ BUILDUPDATEEXPR(h, `a`)
    If *exist* Then Return (true, CONVERT(`type`, *update*))
  Else If n matches "`v = a + b`" Then
    (*exist*, *update*) $\leftarrow$ BUILDUPDATEEXPR(h, `a`)
    If *exist* Then Return (true, *update* + `b`),
    (*exist*, *update*) $\leftarrow$ BUILDUPDATEEXPR(h, `b`)
    If *exist* Then Return (true, *update* + `a`)
  Else If n matches "`v = loop`ϕ`(a, b)`" Then $ln \leftarrow$loop of n
    (notice `a` is defined outside ln and `b` is defined in ln)
    If `a` is defined outside the loop of h Then Return (false, $\perp$)
    $s \leftarrow$ APPLY(ln, ANALYZEEVOLUTION(ln, n),
                NUMBEROFITERATIONS(ln))
    If s matches "`a + t`" Then
      (*exist*, *update*) $\leftarrow$ BUILDUPDATEEXPR(h, `a`)
      If *exist* Then Return (*exist*, *update* + t)
  Else If n matches "`v = cond`ϕ`(a, b)`" Then
    (*exist*, *update*) $\leftarrow$ BUILDUPDATEEXPR(h, `a`) If *exist* Return (true, $\top$)
    (*exist*, *update*) $\leftarrow$ BUILDUPDATEEXPR(h, `b`) If *exist* Return (true, $\top$)
  Return (false, $\perp$)

Figure 4.14: SSA walker: reconstructs symbolic update expressions from a three-address SSA code.

levels. The recursively defined expression is searched and reconstructed from the low level three-address SSA representation using BUILDUPDATEEXPR. This algorithm is presented in Figure 4.14 and it corresponds to a depth-first search algorithm in the SSA graph with each step composed of a look-up of an SSA definition, and then followed by a recursive call of the search algorithm on the symbolic operands. The search halts when the starting loopϕ node is

Algorithm: InstantiateEvolution(*trec*, l)
Input: *trec* a symbolic TREC, l the instantiation loop
Output: an instantiation of *trec*
  If *trec* is a constant c Then Return c
  Else If *trec* is a variable v Then
    If v has not been instantiated Then
      Mark v as instantiated and Return AnalyzeEvolution(l, v)
    Else v is in a mixer structure, Return $\top$
  Else If *trec* is of the form $\{e_1, +, e_2\}_x$ Then
    $i_1 \leftarrow$ InstantiateEvolution(e_1, l)
    $i_2 \leftarrow$ InstantiateEvolution(e_2, l)
    Return $\{i_1, +, i_2\}_x$
  Else If *trec* is of the form $(e_1, e_2)_x$ Then
    $i_1 \leftarrow$ InstantiateEvolution(e_1, l)
    $i_2 \leftarrow$ InstantiateEvolution(e_2, l)
    Return Unify$((i_1, i_2)_x)$
  Else Return $\top$

Figure 4.15: A possible filter function.

reached. When analyzing an assignment whose right-hand side is a sum, the search algorithm examines the first operand, and if the starting loopϕ node is not reachable through this path, it examines the second operand. When one of the operands contains a path to the starting loopϕ node, the other operand of the sum is added to the update expression, and the result is propagated to the lower search steps together with the reconstructed update expression. If the starting loopϕ node cannot be found by depth-first search, i.e., when BuildUpdateExpr returns (false, $\perp$), the definition does not belong to a cycle of the SSA graph: a peeled TREC is returned.

The overall effect of an inner loop may only be computed when the exit value of the variable is a function of the entry value. In such a case, the whole loop is behaving as a macro-increment operation. The function NumberOfIterations computes the number of iterations of the loop by solving a constraint system. As a practical implementation, one can choose the Omega solver [Pug92], but it is also possible to use a solver restricted to univariate affine constraint systems for avoiding any exponential behavior. Then, Apply is used to evaluate the overall effect of the inner loop. Apply implements the efficient evaluation scheme for MCR based on Newton interpolation series presented in Section 4.2. Once the overall effect of an inner

loop on a scalar variable has been computed, the information is propagated after the loop, extending the limits of a classical constant propagation engine after loop structures.

Finally, in the last pattern of AnalyzeEvolution and BuildUpdateExpr, "`v = cond`ϕ`(a, b)`", it is possible to plug the TREC envelope extension by defining the Meet operation.

InstantiateEvolution substitutes symbolic parameters in a TREC. It computes their statically known value, i.e., a constant, a periodic function, or an approximation with intervals, possibly triggering other computations of TREC in the process. The call to InstantiateEvolution is postponed until the end of the depth-first search, ensuring termination of the recursive nesting of depth-first searches, and avoiding early approximations in the computation of update expressions. Combined with the introduction of symbolic parameters in the TREC, postponing the instantiation alleviates the need for a specific ordering of the computation steps.

The termination and complexity of this algorithm are presented in the next subsection, then the algorithm is illustrated with two examples in Section 4.6. The next chapter presents several applications of this static analyzer, and Section 4.7.2 provides experimental results, on standard benchmark programs, of the analyzer that has been integrated in GCC.

4.5 Termination and Complexity of the Algorithm

When analyzing the code of the algorithm, we can briefly sketch its call graph, as shown in Figure 4.11. The algorithm is initiated by a call to AnalyzeEvolution, then finishes with the analysis of all the symbols left in the representation, by calling InstantiateEvolution.

An overview of the ideas that lead to the proof of the termination consists in remarking that:

- AnalyzeEvolution does not analyze twice the same variable, because after each complete analysis, the evolution is stored in a database that is checked on entry of AnalyzeEvolution,

- InstantiateEvolution does not instantiate twice the same variable, otherwise a mixer is detected, and the recursion is stopped either by returning $\top$ as in Figure 4.15, or by translating the mixer in an appropriate abstraction, like a periodic function,

- BUILDUPDATEEXPR terminates once it has reached its halting loopϕ node, or once it has walked over all the SSA edges connected to the starting loopϕ node, in the limits of the analyzed loop.

Consequently, in the worst case, the analyzer stops after having analyzed once all the variables of the program. For giving an idea of the worst case complexity of the analyzer, we will describe with more details the termination process. We will consecutively consider the worst case complexity of each of the building blocks of the algorithm. We deduce from this the overall complexity of the algorithm in the worst case, then the termination of the algorithm in terms of number of basic operations. Figure 4.16 sketches the computational patterns behind each of the components of the algorithm.

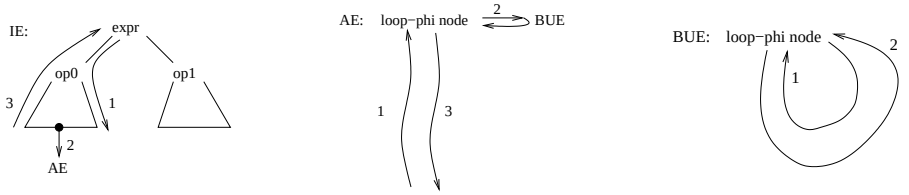


Figure 4.16: Computational patterns of INSTANTIATEEVOLUTION (IE), ANALYZEEVOLUTION (AE), and BUILDUPDATEEXPR (BUE).

Complexity of InstantiateEvolution. The worst case for INSTANTIATEEVOLUTION corresponds to an expression with n operands, and among them appear all the SSA names defined in the program. Let m stand for the number of SSA names in the program. In the worst case $n > m$. The total cost of instantiating such an expression involves a recursive visit of each leaf: that produces n steps, then for each leaf, a call to ANALYZEEVOLUTION. The total number of operations is equal to n decompositions of the given expression, n calls to ANALYZEEVOLUTION, followed by $n - 1$ folds of the obtained subexpressions: the total amounts to $3n - 1$ basic operations.

Complexity of AnalyzeEvolution. The cost of ANALYZEEVOLUTION for a constant is equal to 1. For an SSA name, the cost is equal to the look-up in the database plus, when the scalar variable was not yet analyzed, the cost of its analysis. Because the first part of this algorithm consists in walking up the definitions to known values or to loop-phi nodes, the algorithm may end as soon as all the needed scalar definitions have values already computed.

In the worst case, the total number of steps is equal to $5m$: at each analysis of an SSA name there are at most 3 reads in the database: for

$a = b$ *op* c, a read for a, then supposing that the variable has not yet been analyzed, a read for each of the operands, then a fold operation on the TREC of the operands, and finally a write of the result in the database.

Because in the worst case $n > m$, the difference $n - m$ corresponds to the number of queries from InstantiateEvolution that hit the cached value in the database. Thus, the total number of basic operations for AnalyzeEvolution is equal to $4m + n$.

When AnalyzeEvolution ends on a loopϕ node whose evolution is not yet analyzed, AnalyzeEvolution calls BuildUpdateExpr.

Complexity of BuildUpdateExpr. BuildUpdateExpr is called only from AnalyzeEvolution, when analyzing a loopϕ node. The loopϕ edge exiting the loop is left under a symbolic form, while the edge pointing in the loop is the one followed by BuildUpdateExpr in a depth first search order until the starting loopϕ node is reached. The number of operations triggered by one call to BuildUpdateExpr is equal to the number of edges explored during this depth first search. Once all the paths reachable by following SSA edges in the analyzed loop are explored and the halting loopϕ node is still not found, BuildUpdateExpr ends by returning a peeled TREC.

In the particular case where the SSA edges enter an inner loop, the analysis of the definition in the inner loop is triggered. But because we are computing the total number of steps in the worst case, we have already counted these definitions in the input expression to InstantiateEvolution. Thus, we can consider all these parts already analyzed, and their cost to the BuildUpdateExpr function is equal to a read in the database, if we count the cost of computing the number of iterations for all the loops separately.

BuildUpdateExpr is called only on the loopϕ nodes not yet analyzed, in other words, once per loopϕ node. Thus, in the worst case, the overall complexity of BuildUpdateExpr is equal to $\sum_{i \in \text{loop}\phi} e_i$, where e_i is the number of SSA edges reachable from the analyzed loopϕ , not exiting the loop.

Cost of the Whole Algorithm. Putting all together, we obtain the following worst case complexity:

$$4n - 1 + 4m + \sum_{i \in \text{loop}\phi} e_i + l$$

- n is the number of basic components in TREC to be instantiated,
- m is the number of SSA names in the program,

- e_i is the number of SSA edges reachable from the analyzed loopϕ, and not exiting the loop,
- l is the number of steps required to solve the constraint systems for determining the number of iterations for all the loops. l is linear if the solver is restricted to uniquely deal with univariate affine evolutions. In the case of the Omega solver [Pug92], l is exponential in the worst case.

If the algorithm used for computing the number of iterations is reduced to solving only a single affine equation, the complexity of the scalar evolutions analysis is in $\mathcal{O}(n)$.

We have proved that the algorithm is terminating on any input SSA representation, and we have analyzed the worst case complexity of the analysis algorithm. The overall cost of the analyzer highlights its structure: it is composed of two preprocessing passes followed by the analysis of the loopϕ nodes. The complexity of the algorithm depends on the quality of the expected answer: using an exact solver for constraint systems might not be practical for production compilers. However, in development mode, the plan is to use exact solvers for improving the overall quality of the compiler by assessing regressions with respect to an optimal behavior.

4.6 Application of the Analyzer to the Introductory Examples

We will take the first two introductory examples from Figure 4.3 and Figure 4.4, and will observe the behavior of the analysis algorithm while analyzing these programs. In addition to clarifying the depth-first search and instantiation phases of the algorithm, this will exercise the recognition of polynomial and multivariate scalar evolutions.

First Example. The depth-first search is best understood with the analysis of `c = loop`$_1\phi$`(a, f)` in the first example. The SSA edge of the initial value exits the loop, as represented in Figure 4.17.(1). Here, the initial value is left in a symbolic form, but GCC would replace it by 3 through constant propagation.

To compute the parametric evolution function of `c`, the analyzer starts a depth-first search algorithm, as illustrated in Figure 4.17.(2). The update edge `c`→`f` is followed to the definition of `f` in the loop body: assignment `f = e + c`. The depth-first algorithm follows the first operand, `f`→`e`, reaching

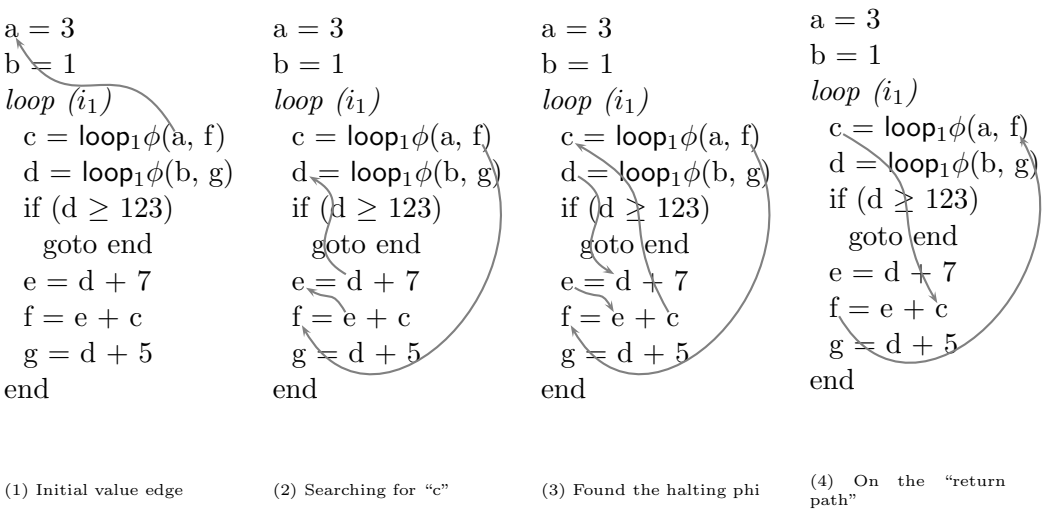


Figure 4.17: Application to the first example

the assignment `e = d + 7`, and finally follows the edge `e`→`d` that leads to a loopϕ node of the same loop. Since this is not the loopϕ node from which the analyzer has started the depth-first search, the search continues on the other operands that were not yet analyzed: back on `e = d + 7`, operand `7` is a scalar and there is nothing more to do, then back on `f = e + c`, the edge `f`→`c` is followed to the starting loopϕ node, as illustrated in Figure 4.17.(3).

At this point, the analyzer has found the strongly connected component that corresponds to the path of iterative updates. Following this path in execution order, as illustrated in Figure 4.17.(4), the analyzer builds the update expression as an aggregation of the operands that are not on the updating path: in this example, the update expression is just `e`. As a result, the analyzer assigns to the definition of `c` the parametric evolution function $\{\mathtt{a}, +, \mathtt{e}\}_1$.

The instantiation of $\{\mathtt{a}, +, \mathtt{e}\}_1$ starts with the substitution of the first operand: `a` = 3, then the analysis of `e` is triggered. First the assignment `e = d + 7` is analyzed, and since the evolution of `d` is not yet known, the edge `e`→`d` is taken to the definition `d = loop`$_1\phi$`(b, g)`. Since this is a loopϕ node, the depth-first search algorithm is used as before and yields the evolution function of `d`, $\{b, +, 5\}_1$, and after instantiation, $\{1, +, 5\}_1$. Finally the evolution of `e = d + 7` is computed: $\{8, +, 5\}_1$, and replacing `e` with its evolution finishes the instantiation of the TREC of `c` that yields $\{3, +, 8, +, 5\}_1$.

Second Example. We now will compute the evolution of `x` in the second example, Figure 4.4, to illustrate the recognition of multivariate induction

variables and the computation of the trip count of a loop. The first step consists in following the SSA-edge to the definition of `x`. Consider the right-hand side of the definition: since the evolution of `e` along loop 1 is not yet analyzed, the edge `e`→`d` is followed to its definition in loop 2, ending on the definition of a loopϕ node.

At this point, `d` is known to be updated in loop 2. The initial value `c` is kept under a symbolic form, and the iteration edge `d`→`e` is followed in the body of loop 2. The depth-first search algorithm starts from right-hand side of the assignment `e = d + 1`: the edge `e`→`d` is followed to the loopϕ node from which the search has started. Back on the path `d`→`e`→`d`, the analyzer gathers the evolution of `d` along the whole loop, an increment of 1, and ends on the following symbolic TREC: $\{\mathtt{c}, +, 1\}_2$.

From the evolution of `d` in the inner loop, the analyzer determines the overall effect of loop 2 on `d`, that is the evaluation of function $f(i) = \mathtt{c} + i$ for the number of iterations of loop 2. Fortunately, the exit condition is the simple expression `t>=9`, and the TREC for `t` (or `d - c`) is $\{0, +, 1\}_2$, an affine (non-symbolic) expression. It comes that 10 iterations of loop 2 will be executed for each iterations of loop 1. Calling Apply(2, $\{\mathtt{c}, +, 1\}_2$, 10) yields the overall effect `d = c + 10`.

The analyzer does not yet know the evolution function of `c`, and consequently it follows the SSA-edge to its definition: `c = loop`$_1\phi$`(a, x)`. Since this is a loopϕ node, the analyzer must determine its evolution in loop 1. The edge to the initial value is ignored, and the update edge is taken, searching for a path from `c` to itself.

First, edge `c`→`x` leads to the statement `x = e + 3`, then following the SSA-edge `x`→`e`, ending on a statement of the loop 2. Again, edge `e`→`d` is followed, ending on the definition of `d` that has already been analyzed: $\{\mathtt{c}, +, 1\}_2$. The depth-first search selects the edge `d`→`c`, associated with the overall effect statement `d = c + 10` that summarizes the evolution of the variable in the inner loop. Finally, the starting loopϕ node `c` is reached. From this point, the path is walked back gathering the stride of the loop: 10 from the assignment `d = c + 10`, then 1 from the assignment `e = d + 1`, and 3 from the last assignment on the return path. The symbolic TREC of `c` has been computed: $\{\mathtt{a}, +, 14\}_1$.

The last step consists in propagating this information from the loopϕ node of `c` to the node where the computation has started: `x`. Back from `c` to `d`, the TREC for `d` can partially be instantiated: $d \rightarrow \{\{\mathtt{a}, +, 14\}_1, +, 1\}_2$. Then back to `e = d + 1`, $e \rightarrow \{\{\mathtt{a}+1, +, 14\}_1, +, 1\}_2$; and finally back to `x = e + 3`, $x \rightarrow \{\mathtt{a}+14, +, 14\}_1$. A final instantiation yields $x \rightarrow \{17, +, 14\}_1$

As it can be seen in these examples, the scalar evolution information gathered on demand is reused several times, and for this reason it is stored

in a cache for avoiding the computation in later queries. This is an important practical design because the scalar evolution information is used by all the loop nest optimizers, from the scalar variable optimizations to the data dependence analyzers whose results are later used in auto-vectorization or loop transforms.

A drawback of using a database is that a part of the information might be invalidated by the loop optimizers: when one of the variables is removed or renamed in the SSA graph, all the scalar evolutions that contain this name have to be invalidated and analyzed again. In practice, it is faster to erase all the results and to start the analysis again. This is not a final result, and we probably will use incremental updates for saving results more difficult to obtain, as for example the results of an interprocedural extension of the analyzer.

4.7 Affine loopϕ Optimizations in GCC

In this section, we survey several applications that use the information extracted under the form of affine evolution functions. We will analyze the impact of the extracted information on the transformation framework of GCC.

4.7.1 Code Transformation Frameworks in GCC

Several transformation passes that use the scalar evolutions analysis have been integrated in GCC: we quickly survey these frameworks and then we will see the importance of the static analysis results in these passes.

Induction Variable Optimizations. Based on the scalar evolutions analysis, several scalar optimizations have been contributed by Zdeněk Dvořák from SuSE: strength reduction, induction variable canonicalization and elimination, loop invariant code motion [ASU86] enabled by default with the option `-O2`.

Data Dependence Analysis. For the development of classical loop optimization techniques, I have implemented several classical data dependence tests on top of the static analysis of scalar evolutions. The dependence tests are organized from the cheapest ones, such as the gcd test, to more complete tests, as the extended Banerjee test and the Omega test [Pug92] that solves affine systems of constraints.

Vectorization. The data dependence analyzer is used in the vectorization phase, targeting the SIMD extensions of modern microprocessors. In order to use the vector units, the "simdization" pass [EWO04] recognizes loop patterns that can be rewritten using SIMD instructions for Altivec, SSE or MMX. This pass is enabled with the option `-ftree-vectorize` and has been contributed by Dorit Nuzman [Nai04, NZ06, NH06, NRZ06] from IBM Haifa.

Selection of SIMD instructions require more efforts than tagging parallel loops. It requires accurate loop-trip count information and well behaved stride array access patterns. The scalar evolutions algorithm can provide some of the required information, in addition to the detection of loop-carried dependences. We have also begun the implementation of the fission-based Kennedy and Allen algorithm [AK87, DRV00] to extract more vectorizable loops from reduced dependence graphs.

Linear Loop Transformations. Daniel Berlin from IBM Research and I have contributed the linear loop transformation framework [BEP04, LP94]. This framework enables a set of classical transformations applicable to perfect loop nests. However for the moment the only loop transformation that contains the static profitability analysis is the loop interchange transformation. Even though the linear loop transformation framework provides a simple interface for translating loop nests, the full power of the framework is not used: more advanced static analysis algorithms have to be implemented for enabling more transformations.

Value Range Propagation. A pass of value range propagation [Pat95] enabled by default at `-O2` has been contributed by Diego Novillo from Red Hat. This value range propagation is based on a generic propagation engine that was described in [Nov05]. This propagation engine is generic enough to be used not only for the propagation of value ranges, but also for other kind of abstractions. We have planned to use this propagation engine to improve the precision of the data dependence analysis in interprocedural mode by propagating accessed array regions represented by sets of affine constraints.

4.7.2 Empirical Study

To show the robustness and language-independence of the implementation of the scalar evolution analyzer integrated in GCC, and to evaluate the accuracy of the algorithm, the analyzer is asked to determine a compact representation of all variables defined by loopϕ nodes in the SPEC CPU2000 [Spe00] and JavaGrande [Jav00] benchmarks. We have also included the total number of

loops per benchmark, together with the results for the static analysis of the number of iterations for all the loops, as the information provided by the scalar evolutions analyzer depends on the precision of the results obtained from the analyzer of the number of iterations. Then, for determining the effect of the scalar evolutions information on the loop nest optimizer, we have completely disabled the scalar evolutions analyzer and measured the performance loss with respect to the same unmodified version of GCC.

Computation of Scalar Evolutions. This first experiment is based on *gcc 4.2.0 20060908 (experimental)*, in which the code for classifying the results of the scalar evolutions analysis has been enabled. Figure 4.18 summarizes the results of this experiment: affine univariate variables are very frequent because well structured loops are most of the time using simple constructs, affine multivariate are also quite frequent because they are used for iterating over multi dimensional arrays. As one could expect, difficult to understand, or even to read, constructs such as polynomials of degree greater or equal to two occur very rarely: the analyzer detected only eight occurrences in SPEC CPU2000, and none in JavaGrande. For the `254.gap` benchmark, that contains the code for an interpreter in group theory computations, the scalar evolutions analyzer detects seven polynomials of degree two. One of the explanations might be that the author of this benchmark has used differentiation mechanisms for computing discrete successive values for polynomials of degree two. The "Cast" column contains the number of evolutions containing cast operations. These evolutions potentially represent wrap around zero evolutions, as the static information is not enough precise to prove their non wrapping behavior. The "$\top$" column gathers all the evolutions that were discarded as too difficult to handle by the analyzer: containing exponentials, evolutions depending on values instantiated only at run time, etc. A column for other kinds of evolutions, such as unfolded additions (e.g. "a + b") and symbolic parameters, is not represented in Figure 4.18, and explains why the sum per line does not match 100% of the number of scalar evolutions presented in the "Scevs" column.

Loop Trip Count. The last four columns in Figure 4.18 show the precision of the detector of the number of iterations. For the moment, only the single-exit loops are exactly analyzed, excluding a big number of loops that contain irregular control flow (probably containing exception exits) such as in the case of Java programs. The effectiveness of the loop transforms is reduced because a large number of loops are not correctly analyzed. In some cases an approximation of the loop count can enable aggressive loop trans-

CINT2000	AU	AM	G	Cast	$\top$	Scevs	$\top_l$	Sym	Stat	$\leq$	Loops
164.gzip	14.03	0.09	0	2.34	81.50	3421	25.67	25.22	16.21	53.60	222
175.vpr	21.27	0.02	0	2.49	73.95	10523	37.90	20.76	4.44	66.33	496
176.gcc	6.41	0.06	0	0.83	91.40	76424	29.18	7.83	9.84	37.93	3385
181.mcf	9.70	0	0	0.59	88.52	845	26.38	19.44	1.39	33.33	72
186.crafty	5.92	0.18	0	1.07	90.24	12940	40.00	7.17	23.91	44.34	460
197.parser	11.95	0.16	0	1.14	84.29	6686	41.72	7.96	0.39	27.08	779
252.eon	19.05	0	0	0.49	75.51	1013	14.28	18.09	23.80	58.09	105
253.perlbmk	4.52	0	0	2.45	91.80	8912	32.20	13.22	2.03	24.06	295
254.gap	10.30	0.18	0.01	3.41	84.39	54980	33.97	17.08	1.97	40.57	2137
255.vortex	6.32	0	0	4.18	87.73	11566	20.36	4.56	1.22	31.91	329
256.bzip2	17.74	0.43	0	3.50	75.65	2575	32.63	23.15	16.31	60.00	190
300.twolf	9.98	0.35	0	1.48	86.85	27397	61.16	10.48	1.84	43.98	1030
CFP2000	**AU**	**AM**	**G**	**Cast**	$\top$	**Scevs**	$\top_l$	**Sym**	**Stat**	$\leq$	**Loops**
168.wupwise	30.35	3.31	0	1.05	61.50	1904	1.22	82.92	1.22	84.14	82
171.swim	30.44	2.47	0	1.24	59.81	647	3.13	78.12	0	78.12	32
172.mgrid	33.33	22.26	0	0.25	41.52	1599	5.80	63.76	8.70	79.71	69
173.applu	26.88	7.18	0	0.29	53.95	6166	6.00	49.45	31.52	83.15	184
177.mesa	17.11	0.16	0	4.88	73.91	33640	13.54	56.20	5.01	71.67	1137
178.galgel	26.99	3.44	0	1.30	62.40	15567	6.72	72.26	5.35	81.84	804
179.art	31.66	0.30	0	1.75	64.84	1317	28.28	37.37	4.04	59.59	99
183.equake	24.91	0.72	0	0.91	72.53	2079	21.90	8.57	43.80	83.80	105
187.facerec	22.37	4.03	0	0.99	66.45	7155	7.01	71.96	3.27	86.91	214
188.ammp	6.69	0.43	0	0.54	90.05	11930	9.29	27.68	2.37	45.71	549
189.lucas	12.65	1.35	0	0.36	80.50	5776	11.00	45.87	33.94	90.82	109
191.fma3d	9.56	0.15	0	3.91	83.09	80356	17.75	28.71	40.87	73.62	2828
200.sixtrack	10.59	1.06	0	0.22	85.75	93258	23.38	48.06	9.09	59.33	1783
301.apsi	31.15	5.62	0.01	0.82	55.65	8739	3.88	71.05	2.33	74.93	387
JavaGrande	**AU**	**AM**	**G**	**Cast**	$\top$	**Scevs**	$\top_l$	**Sym**	**Stat**	$\leq$	**Loops**
section1	0.31	2.67	0	0	96.82	5772	39.05	23.07	0	39.64	169
section2	3.32	11.63	0	0	82.38	2226	2.83	1.89	0	48.11	106
section3	0.25	5.01	0	0	92.80	6848	4.07	0	0.58	33.72	172

Figure 4.18: Scalar induction variables and loop trip count in SPEC CPU2000 and JavaGrande benchmarks. Percentage of scalar evolutions "Scevs" classified into: "AU" affine univariate, "AM" affine multivariate, "G" polynomials of degree greater or equal to two, "Cast" evolutions modified by a cast operation, and "$\top$" undetermined evolutions. The second half of the table classifies in column "$\top_l$" the percentage of loops for which the number of iterations cannot be determined, column "Sym" represents the percentage of loops for which a symbolic number of iterations exists, "Stat" the percentage of loops for which the number of iterations has been statically proven to be a constant integer, "$\leq$" the percentage of loops for which an integer upper bound has been computed, and "Loops" the number of natural loops.

CINT2000	all	+scev	vect	+scev	interch	+scev	ivopts	+scev	ivcanon	+scev	prefetch	+scev	vrp	+scev
164.gzip	1215	1228	1221	1223	1222	1219	1220	1234	1205	1239	1200	1222	1203	1222
175.vpr	1036	1036	1096	1090	1092	1091	1086	1093	1087	1085	1029	1039	1086	1091
181.mcf	603	604	603	605	608	601	606	604	606	604	604	605	606	604
186.crafty	2189	2163	2195	2188	2140	2147	2135	2128	2127	2138	2184	2189	2196	2192
197.parser	897	891	920	921	921	918	921	915	922	919	895	893	920	921
253.perlbmk	1491	1479	1544	1546	1537	1544	1538	1545	1541	1542	1493	1500	1544	1542
254.gap	1186	1187	1191	1162	1192	1186	1188	1178	1178	1172	1194	1190	1173	1179
256.bzip2	1135	mis	1149	mis	1150	1149	1151	1156	1158	1157	1140	1140	1152	1153
300.twolf	1171	1182	1236	1238	1249	1244	1245	1248	1240	1228	1149	1161	1244	1242
CFP2000	**all**	**+scev**	**vect**	**+scev**	**interch**	**+scev**	**ivopts**	**+scev**	**ivcanon**	**+scev**	**prefetch**	**+scev**	**vrp**	**+scev**
168.wupwise	1241	1430	1276	1274	1275	1277	1275	1416	1272	1273	1237	1238	1275	1275
171.swim	1100	1329	1099	1099	1102	1325	1104	1103	1105	1103	1104	1104	1104	1103
172.mgrid	727	837	786	727	801	800	800	1031	800	801	727	726	799	799
173.applu	775	850	772	844	776	857	776	910	840	772	848	789	864	786
177.mesa	1633	1066	1649	1680	1637	1685	1638	1636	1690	1638	1280	1596	1662	1680
178.galgel	915	945	883	898	900	898	892	942	899	893	912	911	904	896
179.art	1070	1081	1079	1083	1095	1066	1041	1063	1074	1074	1042	1077	1085	1096
183.equake	1111	1118	1097	1099	1112	1100	1100	1116	1095	1101	1116	1113	1107	1102
187.facerec	933	978	952	954	954	955	953	983	952	950	925	929	956	956
188.ammp	1317	1329	1343	1331	1353	1344	1343	1318	1343	1339	1316	1311	1348	1345
189.lucas	1119	1152	1114	1122	1118	1121	1114	1149	1118	1116	1117	1119	1117	1119
191.fma3d	982	983	955	956	960	948	955	961	986	992	961	960	947	936
200.sixtrack	618	634	654	654	655	653	654	646	656	652	618	615	655	653
301.apsi	1122	1205	1183	1194	1190	1191	1189	1261	1192	1190	1126	1124	1197	1199

Figure 4.19: Impact of the scalar evolution analyzer of GCC version 4.1 from 2006-12-18 on code transformations for the SPEC CPU2000 benchmarks on AMD64. First two columns "all +scev" represent the overall effect of the compiler, with the following flags "-O3 -msse2 -ftree-vectorize -ftree-loop-linear -fivopts -ftree-loop-ivcanon -fprefetch-loop-arrays -ftree-vrp", first with the scalar evolution analyzer disabled then with the analyzer enabled. The values correspond to the SPEC CPU2000 normalized numbers: reference run time divided by the measured run time multiplied by 100, high values corresponding to better performance results. Using the same presentation pattern, next columns give more details on the effect of the scalar evolution analyzer on a single code transformation: "vect" represent the vectorization pass, "interch" the loop interchange, "ivopts" the induction variable optimizations, "ivcanon" the induction variable canonicalization, "prefetch" the array prefetching pass, and "vrp" the value range propagation pass.

formations as is the case of the 171.swim test in SPEC CPU2000: the size of data accessed in the loop is used to provide an upper bound estimation of the number of iterations, allowing the dependence analyzer to correctly refine the dependence relations, and the loop interchange to be performed. Further refinements of this analyzer will either provide more hints for approximating the number of iterations, or use an integer programming solver, such as Pugh's Omega solver [Pug92], or a polyhedral library like the Polylib [LW97, Cla96a], for precisely computing or approximating the number of iterations in multiple-exit loops.

Impact of Scalar Evolutions Analysis on GCC. For evaluating the importance of the induction variable analysis on existing optimizations in GCC, I used two compilers based on *gcc version 4.1.0 20051104 (experimental)* with the following options "-O3 -msse2 -ftree-vectorize -ftree-loop-linear": the peak compiler is the compiler without modifications; in the base compiler I disabled the analysis of induction variables, by making the analyzer systematically return "don't know" for every query.

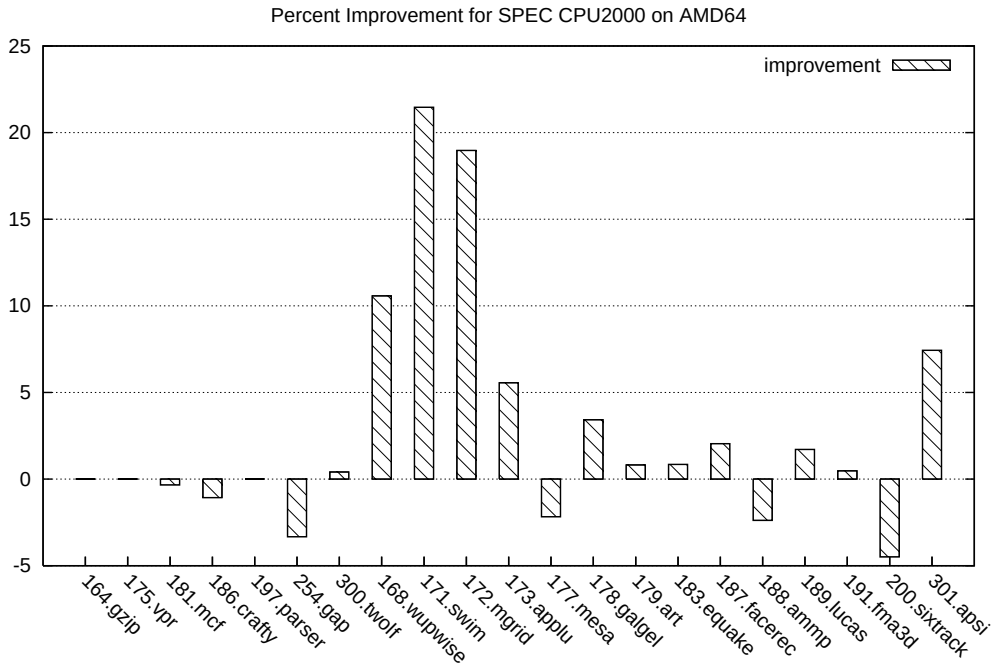


Figure 4.20: Percent speed up of run time for SPEC CPU2000 on AMD64.

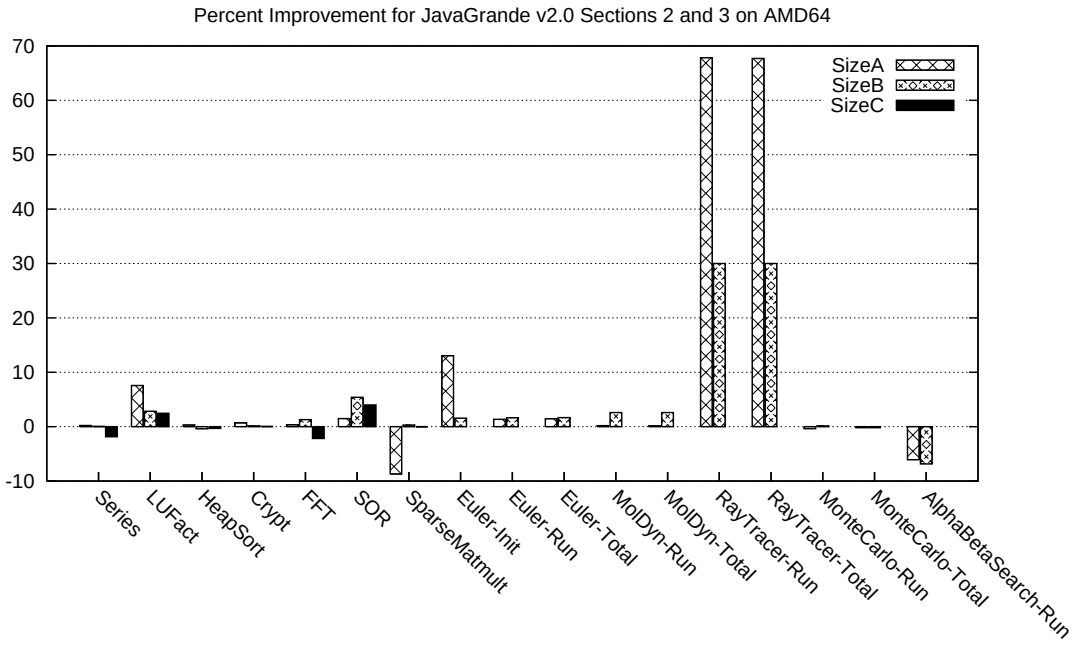


Figure 4.21: Percent speed up of run time for JavaGrande on AMD64.

Figures 4.20 and 4.21 present the percent improvement for execution time for the SPEC CPU2000 and for the JavaGrande benchmarks on an AMD Athlon(tm) 64 Processor 3700+ with 128 Kb of L1 cache, 1024 Kb of L2, and 1 GB of RAM, on SuSE with a Linux kernel 2.6.13. Figure 4.19 illustrates the impact of the scalar evolution analyzer separately on each code transformation that potentially uses this information.

The *171.swim* benchmark is written in FORTRAN. The SPEC CPU2000 developers have included a verification loop that is iterating over the data of a table following the lines. Iterating following the lines of a multi dimensional array in FORTRAN is not as efficient as iterating over the columns elements: accessing to two adjacent line elements can potentially produce a cache miss, a long delay needed to fetch the data from the main memory into the caches, as the successive accessed elements are separated by the number of elements contained in a column that might be larger than the size of the cache line containing the first accessed element. This cache miss phenomenon is not occurring when the accessed elements are adjacently stored in memory. This kind of anomaly is not language specific, as it can as well occur in other languages. The problem is detected in a language-independent way in the

loop nest optimizer by computing the data access strides for each array. The problem is solved by modifying the iteration order over the table, reducing the number of cache misses, that improves the overall execution time of the loop. In the case of the *171.swim* benchmark we can see an improvement of 21% by enabling the loop interchange transformation.

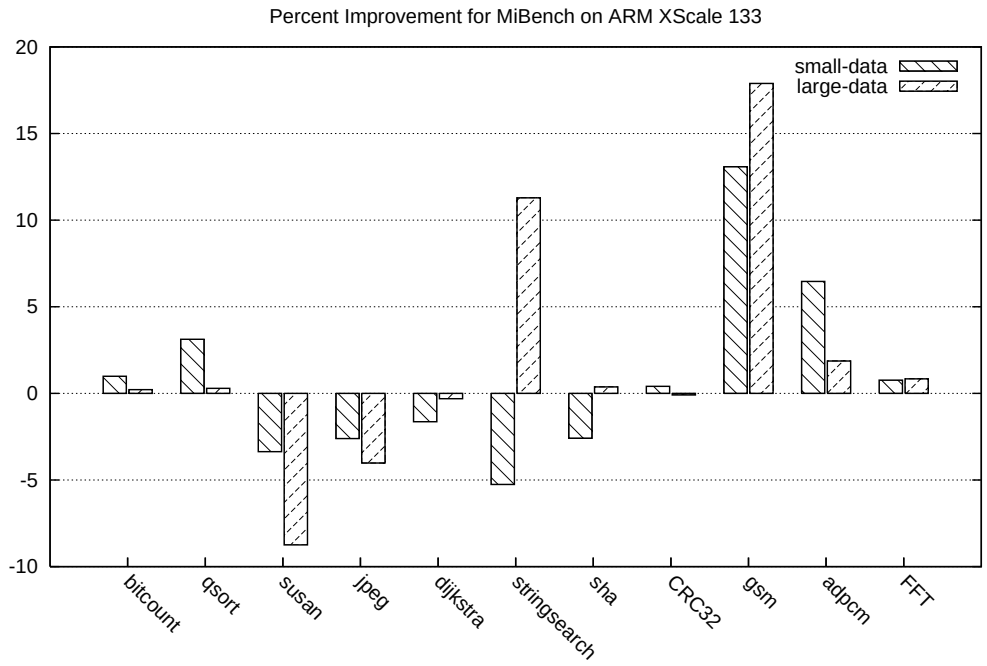


Figure 4.22: Percent speed up of run time for MiBench on ARM.

Figure 4.22 presents the percent improvement of execution time for a part of the MiBench on an ARM XScale-IXP42x processor at 133 MHz with 32 Mb of RAM on Debian with Linux kernel 2.6.12. It is possible to remark that knowing more information about the compiled program can produce slower executing programs, as some of the transformations are applied systematically without statically analyzing the effect of the transformation on the target machine. However, it is possible to remark that overall, the information provided by the scalar evolutions analysis is crucial for the validation of aggressive code transformations.

Compilation Time. Looking at compilation time issues, a possible technique for stressing and measuring the compilation time of the scalar evolu-

tions analyzer is to look at an optimizer that uses this analyzer: the vectorizer is based on a pattern matching of the instructions contained in loops, and on the data dependence information. The vectorizer uses both the scalar evolutions analyzer and the data dependence analyzer. The results of the data dependence tests are obtained on demand for a loop nest, and thus, the analysis can be restricted to the loop to be optimized. Furthermore, the dependence tests are ordered such that the fastest tests are executed first, then more expensive analyses are triggered if the previous ones fail. In order to show the effectiveness of the analysis framework, we have measured the compilation time of the vectorization pass. For the SPEC CPU2000 benchmarks, the vectorization pass does not exceed 1 second per compiled file, nor 5 percent of the compilation time per file, showing that the dependence analyzer is fast in practice.

These experiments show that the scalar evolutions analysis framework and the data dependence analysis that I integrated to GCC are robust for a large set of benchmarks written in several programming languages (C, C++, Java, Fortran) and have a low overhead compilation time. However several experiments have shown some degradations of the execution time after transformations enabled by the scalar evolutions analysis results. These cases have to be carefully analyzed and improved by integrating more static analysis results on the profitability of code transformations.

4.8 Conclusion

In the first part of this chapter we have seen an informal description of the scalar evolutions analyzer that we have formally described in the previous chapter. In the current chapter we focused on the description of this analyzer from an engineering point of view: this presentation is suited to be integrated in industrial compilers. The analyzer extracts from the SSA language several more abstract intermediate representations: the TREC are instantiated to MCR, from which affine scalar evolutions are selected. We have also analyzed the complexity of this analyzer, a linear time in terms of the number of SSA declarations in the program, and on the time needed to compute the number of iterations in the analyzed loops. This imperative version of the scalar evolution analyzer has been integrated in the main versions of GCC starting with version 4.0. We have seen the results of several experimentations that measured the precision of the extracted information, the compile time overhead due to the static analysis, the impact of the scalar evolutions analysis on the GCC's transformation frameworks, and the robustness and efficiency of the implementation.

Chapter 5

Conclusions and Future Work

This thesis contributed to the formal definition of the Static Single Assignment (SSA) form and to the understanding of static program analysis based on the SSA language. An important part of these advances comes from practice, where these ideas have been in use for some time now: this thesis collected and formalized several of these ideas that appeared in the GCC community. Thus, the contributions of this thesis contain extensions to both formal and practical frameworks, as detailed below.

5.1 Contributions to Formal Frameworks

This thesis presents the first denotational semantics of the SSA language. Previous attempts to formalize this ubiquitous compiler representation were not entirely satisfactory for defining and providing correctness proofs for the translation of imperative languages to the SSA form. This thesis provides the definitions of the SSA language, under the form of a syntax and a denotational semantics, and the conversion algorithm from the imperative language IMP to SSA, together with the consistency theorem and its proof. The formalization of this translation process provided some deep insights in the nature of the SSA language: the SSA was misnamed, as it is a language of declarations, while assignments are language constructs that occur uniquely in imperative languages. Indeed, the notion of assignment is only relevant in programming languages that contain the notion of store, that also implies the existence of operations of rewriting the same store location. In the case of the SSA language, the notion of store does not exist, as it is replaced by the notion of declaration, and stream of values.

The values taken by a recursively declared variable can equivalently be viewed as an infinite stream of values, or as an evolving scalar value, justifying

the name that we have commonly used in practice: the scalar evolution of a variable. The expressiveness power of a language of infinite streams is equivalent to primitive recursive functions. The addition of a minimization operation, corresponding to the semantics of the closeϕ element in the SSA language, boosts the expressiveness power of the SSA language to the full rank of partial recursive functions equivalent to Random Access Machines (RAM). The compilation function from the imperative language IMP to the SSA language provides thus another way of converting any RAM program to a set of Kleene's partial recursive functions, thus providing a new proof of Turing's Equivalence Theorem between these two computational models, previously typically proven using simulation.

As an exercise of style, we have seen a possible way to work with the denotational semantics of the SSA in the classical abstract interpretation framework. For illustrating the SSA abstract interpretation framework, we have seen a possible definition of the SSA language in terms of PROLOG predicates, and then we have seen the definition of several abstraction functions that extract abstract views of the program as abstract domains.

We have also seen an imperative description of the static analysis algorithms described earlier in PROLOG. This imperative version of the analysis is the closest to the implementation of this analyzer that I have implemented and integrated to GCC. From a theoretical point of view, the originality of the work on induction variable analysis that I present in this thesis is based on a classical perspective on induction variable analysis: starting from a formal semantics of the source representation, the SSA language, filtering out difficult elements, or selecting only a part of the SSA constructs, produces a subset of the SSA called TREC, from which other abstracting processes translate the TREC into a less expressive representations, such as the affine functions. These abstracting processes are performed on the demand of a user pass of this information, like the data dependence analysis or the vectorizer.

5.2 Contributions to Practical Frameworks

From a practical point of view, this thesis has provided several code contributions to an industrial compiler. Among these contributions, several of the formal ideas developed in the first parts of this thesis have been implemented and integrated in the main versions of the GNU Compiler Collection (GCC) starting with version 4.0.

This thesis has presented an evaluation of the practical contributions: we have seen several code transformation frameworks that have been developed around the information extracted by the scalar evolution analyses. We then

have presented some experiments that show the effectiveness and the impact of the scalar evolutions analysis on the code transformation passes that are using the informations provided by this analyzer. We have seen that the compilation time overhead of the scalar evolutions analysis in the context of a real optimization pass, the vectorizer, is minimal, and is acceptable in an industrial compiler framework. We remarked that the impact of the scalar evolutions analysis on the generated code was not always positive: in several cases the effect of code transformations enabled by a more precise knowledge resulted in a degradation of the execution time. We finally pointed out several directions for improving the quality of the generated code on which we have already started to work: the guideline ideas that we proposed are the implementation of more precise static profitability analyses.

5.3 Future Work

We have left several points for future work, both in the theoretical and in the practical frameworks.

In the theoretical part, we still have not addressed the translation of the SSA language back to the IMP language. This point was not critical for our presentation of the scalar evolution analysis under the framework of abstract interpretation and thus it received less attention, but is nevertheless an important problem for the theoretical description of compilers. We have tried to define this conversion function, but the main difficulty that we have not yet solved is to recover the loop structures from the SSA representation. Indeed the SSA language stores this structure uniquely in the declarations of scalar variables, from which one has to deduce the original or an equivalent loop nesting.

In the practical part, the results of the scalar evolution analysis pave the way to more advanced static analyses, and to more code transformation frameworks. We have several future plans and directions that will improve the quality of GCC's frameworks, and will be further areas of investigation in future research: the GRAPHITE project [PCB+06] aims at extending the existing loop nest optimizer to more operations on polyhedral representation. The automatic parallelization project will provide a code generation framework targeting the OpenMP runtime library, and a project on static machine models aims at describing more precisely the underlying architectures for improving the static profitability analyses. Finally, we think that more profitability analyses will need to be implemented and then tuned. They will have to be built simultaneously with a performance regression testsuite that will ensure the performance enhancement on several different architectures.

Bibliography

[ACF06] P. Amiranoff, A. Cohen, and P. Feautrier. Beyond iteration vectors: Instancewise relational abstract domains. In *Static Analysis Symposium (SAS'06)*, Seoul, Corea, August 2006.

[ADvF01] W. Amme, N. Dalton, J. von Ronne, and M. Franz. SafeTSA: a type safe and referentially secure mobile-code representation based on static single assignment form. In *ACM Symp. on Programming Language Design and Implementation (PLDI'01)*. ACM Press, 2001.

[AK87] J. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. *ACM Trans. on Programming Languages and Systems*, 9(4):491–542, October 1987.

[AK99] Hassan Aït-Kaci. *Warren's Abstract Machine a Tutorial Reconstruction*. `http://www.isg.sfu.ca/~hak/documents/wam.html`, 1999.

[AK02] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan and Kaufman, 2002.

[Ami04] P. Amiranoff. *Analyse de programmes par instances par le biais des transducteurs*. PhD thesis, 2004.

[ASU86] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[AWZ88] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of values in programs. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 1–11, New York, NY, USA, 1988. ACM Press.

[Bac78] John Backus. Can programming be liberated from the von Neumann style? a functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641, 1978.

[BCHS98] Preston Briggs, Keith D. Cooper, Timothy J. Harvey, and L. Taylor Simpson. Practical improvements to the construction and destruction of static single assignment form. *Software Practice and Experience*, 28(8):859–881, 1998.

[BEP04] Daniel Berlin, David Edelsohn, and Sebastian Pop. High-level loop optimizations for GCC. In *Proceedings of the 2004 GCC Developers Summit*, pages 37–54, 2004. `http://www.gccsummit.org/2004`.

[BM94] Marc M. Brandis and Hanspeter Mossenbock. Single-pass generation of static single-assignment form for structured languages. *ACM Trans. Program. Lang. Syst.*, 16(6):1684–1698, 1994.

[BP99] Gianfranco Bilardi and Keshav Pingali. The static single assignment form and its computation. Technical report, Department of Computer Science, Cornell University, Jul 1999.

[BP03] Gianfranco Bilardi and Keshav Pingali. Algorithms for computing the static single assignment form. *J. ACM*, 50(3):375–425, 2003.

[BWZ94] Olaf Bachmann, Paul S. Wang, and Eugene V. Zima. Chains of recurrences a method to expedite the evaluation of closed-form functions. In *Proceedings of the international symposium on Symbolic and algebraic computation*, pages 242–249. ACM Press, 1994.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *4^{th}ACM Symp. on Principles of Programming Languages*, pages 238–252, Los Angeles, California, January 1977.

[CC79] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, 1979. ACM Press, New York, NY.

[CFR+89] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *POPL '89: Proceedings of the 16th ACM*

SIGPLAN-SIGACT symposium on Principles of programming languages, pages 25–35, New York, NY, USA, 1989. ACM Press.

[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 Trans. on Programming Languages and Systems*, 13(4):451–490, October 1991.

[CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *5thACM Symp. on Principles of Programming Languages*, pages 84–96, January 1978.

[Cla96a] P. Clauss. Counting solutions to linear and nonlinear constraints through Ehrhart polynomials: Applications to analyze and transform scientific programs. In *ACM Int. Conf. on Supercomputing*, pages 278–295. ACM Press, 1996.

[Cla96b] Ph. Clauss. Counting solutions to linear and nonlinear constraints through ehrhart polynomials: Applications to analyze and transform scientific programs. In *ICS '96: Proceedings of the 10th international conference on Supercomputing*. ACM Press, 1996.

[CT04] Ph. Clauss and Irina Tchoupaeva. A symbolic approach to bernstein expansion for program analysis and optimization. In *13th International Conference on Compiler Construction, CC 2004*, number 2985 in LNCS. Springer-Verlag, 2004.

[DRV00] A. Darte, Y. Robert, and F. Vivien. *Scheduling and Automatic Parallelization*. Birkhaser, Boston, 2000.

[Dvo] Zdenek Dvorak. [lno] Enable unrolling/peeling/unswitching of arbitrary loops. `http://gcc.gnu.org/ml/gcc-patches/2004-03/msg00212.html`.

[EWO04] Alexandre E. Eichenberger, Peng Wu, and Kevin O'Brien. Vectorization for simd architectures with alignment constraints. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 82–93. ACM Press, 2004.

[Fer04] Jérôme Feret. Static analysis of digital filters. In *European Symposium on Programming (ESOP'04)*, number 2986 in LNCS. Springer-Verlag, 2004. Springer-Verlag.

[Fer05] Jérôme Feret. The arithmetic-geometric progression abstract domain. In *Verification, Model Checking and Abstract Interpretation (VMCAI'05)*, number 3385 in LNCS, pages 42–58. Springer-Verlag, 2005. Springer-Verlag.

[GCC] GCC implementation of "out of SSA". `http://gcc.gnu.org/viewcvs/trunk/gcc/tree-outof-ssa.c`.

[GCC05] The GNU Compiler Collection, 2005. `http://gcc.gnu.org`.

[Gle04] Sabine Glesner. An ASM semantics for SSA intermediate representations. In *Proceedings of the 11th International Workshop on Abstract State Machines*, volume 3052. Springer Verlag, Lecture Notes in Computer Science, Mai 2004.

[GNU] GNU Prolog. `http://pauillac.inria.fr/~diaz/gnu-prolog/`.

[Gor79] Michael J. C. Gordon. *The denotational description of programming languages*. Springer Verlag, 1979.

[Gra91] Philippe Granger. Static analysis of linear congruence equalities among variables of a program. In *TAPSOFT '91: Proceedings of the international joint conference on theory and practice of software development on Colloquium on trees in algebra and programming (CAAP '91): vol 1*, volume 493, pages 169–192, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

[GSW95] M. P. Gerlek, E. Stoltz, and M. J. Wolfe. Beyond induction variables: detecting and classifying sequences using a demand-driven ssa form. *ACM Trans. on Programming Languages and Systems*, 17(1):85–122, January 1995.

[Gur95] Yuri Gurevich. Evolving algebras 1993: Lipari guide. In *Specification and Validation Methods*, pages 231–243. Oxford University Press, 1995.

[Hav93] Paul Havlak. Construction of thinned gated single-assignment form. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, volume 768 of *Lecture Notes in Computer Science*, pages 477–499. Springer, 1993.

[HDE+93] L. Hendren, C. Donawa, M. Emami, G. R. Gao, Justiani, and B. Sridharan. Designing the McCAT compiler based on a family of structured intermediate representations. In *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, number 757 in LNCS, pages 406–420. Springer-Verlag, 1993.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[HP96] M. Haghighat and C. Polychronopoulos. Symbolic analysis for parallelizing compilers. *ACM Trans. on Programming Languages and Systems*, 18(4):477–518, July 1996.

[ICC] Intel compilers. `http://intel.com/`.

[ISO95] Information technology - programming languages - prolog - part 1: General core. ISO/IEC 13211-1. http://www.logic-programming.org/prolog_std.html, 1995.

[Jav00] Java grande forum, 2000. `http://www.javagrande.org`.

[Jon97] Neil D. Jones. *Computability and complexity: from a programming perspective*. MIT Press, Cambridge, MA, USA, 1997.

[Kel95] Richard A. Kelsey. A correspondence between continuation passing style and static single assignment form. *ACM SIGPLAN Notices*, 30(3):13–22, 1995.

[KMZ98] V. Kislenkov, V. Mitrofanov, and E. Zima. Multidimensional chains of recurrences. In *Proceedings of the 1998 international symposium on symbolic and algebraic computation*, pages 199–206. ACM Press, 1998.

[LA04] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *ACM Symp. on Code Generation and Optimization (CGO'04)*, Palo Alto, California, March 2004.

[Lat] Chris Lattner. Scalar evolutions. Personal communication.

[LP94] W. Li and K. Pingali. A singular loop transformation framework based on non-singular matrices. *Intl. J. of Parallel Programming*, 22(2):183–205, April 1994.

[LPS05] Daniel Luna, Mikael Pettersson, and Konstantinos Sagonas. Efficiently compiling a functional language on AMD64: the HiPE experience. In *PPDP '05: Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 176–186, New York, NY, USA, 2005. ACM Press.

[LW97] V. Loechner and D. Wilde. Parameterized polyhedra and their vertices. *Int. J. of Parallel Programming*, 25(6), December 1997. `http://icps.u-strasbg.fr/PolyLib`.

[Mas92] François Masdupuy. Array abstractions using semantic analysis of trapezoid congruences. In *ICS '92: Proceedings of the 6th international conference on Supercomputing*, pages 226–235, New York, NY, USA, 1992. ACM Press.

[Mas93] F. Masdupuy. Semantic analysis of interval congruences. In D. Brner, M. Broy, and I. V. Pottosin, editors, *Intl. Conf. on Formal Methods in Programming and their Applications*, volume 735 of *LNCS*, pages 142–155, Academgorodok, Novosibirsk, Russia, June 1993. Springer-Verlag.

[Mei04] Benoît Meister. *Stating and Manipulating Periodicity in the Polytope Model. Applications to Program Analysis and Optimization.* PhD thesis, Université Louis Pasteur, 2004.

[Mer03] Jason Merill. GENERIC and GIMPLE: a new tree representation for entire functions. In *Proceedings of the 2003 GCC Developers Summit*, pages 171–180, 2003. `http://www.gccsummit.org/2003`.

[Min01] Antoine Miné. The octagon abstract domain. In *AST 2001 in WCRE 2001*, IEEE, pages 310–319. IEEE CS Press, October 2001.

[Moo66] R. E. Moore. *Interval Analysis.* Prentice-Hall, Englewood Cliffs N. J., 1966.

[Muc97] S. S. Muchnick. *Advanced Compiler Design & Implementation.* Morgan Kaufmann, 1997.

[Nai04] Dorit Naishlos. Autovectorization in GCC. In *Proceedings of the 2004 GCC Developers Summit*, pages 105–118, 2004. `http://www.gccsummit.org/2004`.

[NH06] Dorit Nuzman and Richard Henderson. Multi-platform auto-vectorization. In *Proceedings 4th Annual International Symposium on Code Generation and Optimization*. ACM Press, 2006.

[NNH99] F. Nielson, H. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[Nov] Diego Novillo. SIMPLE: A language-independent tree IR. `http://gcc.gnu.org/ml/gcc/2002-01/msg00082.html`.

[Nov03] Diego Novillo. Tree SSA - a new optimization infrastructure for GCC. In *Proceedings of the 2003 GCC Developers Summit*, pages 181–193, 2003. `http://www.gccsummit.org/2003`.

[Nov05] Diego Novillo. A propagation engine for GCC. In *Proceedings of the 2005 GCC Developers Summit*, pages 175–185, 2005. `http://www.gccsummit.org/2005`.

[NRZ06] Dorit Nuzman, Ira Rosen, and Ayal Zaks. Auto-vectorization of interleaved data for SIMD. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM Press, 2006.

[NZ06] Dorit Nuzman and Ayal Zaks. Autovectorization in GCC – two years later. In *Proceedings of the 2006 GCC Developers Summit*, pages 145–158, 2006. `http://www.gccsummit.org/2006`.

[Pap98] Nikolaos S. Papaspyrou. *A Formal Semantics for the C Programming Language*. PhD thesis, National Technical University of Athens, Feb 1998.

[Pat95] Jason R. C. Patterson. Accurate static branch prediction by value range propagation. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 67–78. ACM Press, 1995.

[PCB+06] Sebastian Pop, Albert Cohen, Cédric Bastoul, Sylvain Girbal, Georges-André Silber, and Nicolas Vasilache. Graphite: Polyhedral analyses and optimizations for GCC. In *Proc. of the 4th GCC Developper's Summit*, Ottawa, Canada, June 2006.

[PCC+04] Sebastian Pop, Philippe Clauss, Albert Cohen, Vincent Loechner, and Georges-André Silber. Fast recognition of scalar evolutions on three-address SSA code. Technical Report A/354/CRI,

Centre de Recherche en Informatique (CRI), École des mines de Paris, 2004. `http://www.cri.ensmp.fr/classement/doc/A-354.ps`.

[PCJS06a] Sebastian Pop, Albert Cohen, Pierre Jouvelot, and Georges-André Silber. Denotational semantics for SSA conversion. Technical Report A/379/CRI, Centre de Recherche en Informatique (CRI), École des mines de Paris, 2006. `http://www.cri.ensmp.fr/classement/doc/A-379.pdf`.

[PCJS06b] Sebastian Pop, Albert Cohen, Pierre Jouvelot, and Georges-André Silber. The new framework for loop nest optimization in GCC: from prototyping to evaluation. In *Proc. of the 12th Workshop on Compilers for Parallel Computers (CPC'06)*, A Coruña, Spain, January 2006.

[PCS05] Sebastian Pop, Albert Cohen, and Georges-André Silber. Induction variable analysis with delayed abstractions. In *(HiPEAC'05)*, number 3793 in LNCS, pages 218–232, Barcelona, Spain, November 2005. Springer-Verlag.

[Pug92] W. Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):27–47, August 1992.

[PW76] M. S. Paterson and M. N. Wegman. Linear unification. In *STOC '76: Proceedings of the eighth annual ACM symposium on Theory of computing*, pages 181–186, New York, NY, USA, 1976. ACM Press.

[PW78] M. S. Paterson and M. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16:158–167, 1978.

[Rog87] Hartley Rogers. *Theory of Recursive Functions and Effective Computability*. MIT Press, Cambridge, MA, USA, 1987.

[RWZ88] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 12–27, New York, NY, USA, 1988. ACM Press.

[RZR04] Silvius Rus, Dongmin Zhang, and Lawrence Rauchwerger. The value evolution graph and its use in memory reference analysis.

In *Proceedings of the 2004 Conference on Parallel Architectures and Compilation Techniques*. IEEE Computer Society, 2004.

[Sco82] Dana S. Scott. Domains for denotational semantics. In *Proceedings of the 9th Colloquium on Automata, Languages and Programming*, pages 577–613, London, UK, 1982. Springer-Verlag.

[SJGS99] Vugranam C. Sreedhar, Roy Dz-Ching Ju, David M. Gillies, and Vatsa Santhanam. Translating out of static single assignment form. In *SAS '99: Proceedings of the 6th International Symposium on Static Analysis*, pages 194–210, London, UK, 1999. Springer-Verlag.

[SM76] Christopher Strachey and Robert Milne. *A theory of programming language semantics*. Chapman and Hall, 1976.

[Spe00] Standard performance evaluation corporation, 2000. `http://www.spec.org`.

[SS70] R. M. Shapiro and H. Saint. The representation of algorithms. Technical Report CA-7002-1432, Massachusetts Computer Associates, Feb 1970.

[Sto77] Joseph E. Stoy. *Denotational Semantics: the Scott-Strachey Approach to Programming Languages Theory*. MIT Press, 1977.

[Tar74] R. E. Tarjan. Finding dominators in directed graphs. *SIAM J. Computing*, 3(1):62–89, 1974.

[TP95a] P. Tu and D. Padua. Gated SSA-Based demand-driven symbolic analysis for parallelizing compilers. In *ACM Int. Conf. on Supercomputing*, pages 414–423, Barcelona, Spain, July 1995.

[TP95b] Peng Tu and David Padua. Efficient building and placing of gating functions. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, pages 47–55, 1995.

[vE01] R. A. van Engelen. Efficient symbolic analysis for optimizing compilers. In *Proceedings of the International Conference on Compiler Construction (ETAPS CC'01)*, pages 118–132, 2001.

[vEBS+04] Robert van Engelen, Johnnie Birch, Yixin Shou, Burt Walsh, and Kyle Gallivan. A unified framework for nonlinear dependence testing and symbolic analysis. In *Proceedings of the ACM*

International Conference on Supercomputing (ICS), pages 106–115, 2004.

[Ven02] Arnaud Venet. Nonuniform alias analysis of recursive data structures and arrays. In *Proceedings of the 9th International Symposium on Static Analysis SAS'02*, number 2477 in LNCS, pages 36–51. Springer-Verlag, 2002. Springer-Verlag.

[WCP01] P. Wu, A. Cohen, and D. Padua. Induction variable analysis without idiom recognition: Beyond monotonicity. In *Workshop on Languages and Compilers for Parallel Computing (LCPC'01)*, volume 2624 of *LNCS*, Cumberland Falls, Kentucky, August 2001. Springer-Verlag.

[Wol92] M. J. Wolfe. Beyond induction variables. In *ACM Symp. on Programming Language Design and Implementation (PLDI'92)*, pages 162–174, San Francisco, California, June 1992.

[Wol96] M. J. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.

[WZ91] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Trans. on Programming Languages and Systems*, 13(2):181–210, 1991.

[Zada] F. Kenneth Zadeck. Loop closed SSA form. Personal communication.

[Zadb] F. Kenneth Zadeck. Static single assignment form, 2004 GCC Summit keynote. `http://naturalbridge.com/GCC2004Summit.pdf`.

[Zim01] Eugene V. Zima. On computational properties of chains of recurrences. In *Proceedings of the 2001 international symposium on symbolic and algebraic computation*, pages 345–352. ACM Press, 2001.

Résumé

Le langage d'assignation statique unique, SSA, est l'une des représentations intermédiaires les plus communément utilisées dans les compilateurs industriels. Cependant l'intérêt de la communauté d'analyse statique de programmes est minime, un fait dû aux faibles fondations formelles du langage SSA.

Cette thèse présente une sémantique dénotationelle du langage SSA, permettant des définitions formelles des analyses statiques du langage SSA en se basant sur les méthodes classiques de l'interprétation abstraite. D'un point de vue pratique, cette thèse présente l'implémentation des analyseurs statiques définis formellement dans un compilateur industriel, la Collection de Compilateurs GNU, GCC.
Mots-clés : analyse statique de programmes, langage d'assignation statique unique, SSA, sémantique dénotationelle, GCC, interprétation abstraite

Abstract

The Static Single Assignment (SSA) language is one of the intermediate representations commonly used in industrial compilers. However, there was little interest from the static program analysis community in this intermediate representation due to the weak formal grounds of the SSA.

This thesis presents a denotational semantics of the SSA language, allowing formal definitions of static analyses on the SSA language based on the classical abstract interpretation framework. From a practical point of view, we present the implementation of the formally described static analyses on the SSA in an industrial compiler: the GNU Compiler Collection (GCC).
Keywords: program static analysis, static single assignment language, SSA, denotational semantics, GCC, abstract interpretation