Università
Ca'Foscari
Venezia

Scuola Dottorale di Ateneo
Graduate School

Dottorato di ricerca
in Informatica
Ciclo XXIV
Anno di discussione 2013

Information Extraction by Type Analysis

SETTORE SCIENTIFICO DISCIPLINARE DI AFFERENZA: INF/01
Tesi di Dottorato di Alvise Spanò, matricola 955680

Coordinatore del Dottorato	Tutore del Dottorando
Prof. Riccardo Focardi	Prof. Michele Bugliesi

Author's Web Page: http://www.dsi.unive.it/~spano

Author's e-mail: spano@dsi.unive.it

Author's address:

Dipartimento di Informatica
Università Ca' Foscari di Venezia
Via Torino, 155
30172 Venezia Mestre – Italia
tel. +39 041 2348411
fax. +39 041 2348419
web: http://www.dsi.unive.it

to Aliel, bringer of the storm and riser of the tide.

Abstract

Type checking is widely used for static analysis and code verification. This thesis investigates an alternative use of type reconstruction, as a tool for extracting knowledge from programs written in weakly typed language. We explore this avenue along two different, but related directions.

In the first part we present a static analyzer that exploits typing techniques to extract information from the COBOL source code: reconstructing informative types is an effective way for automatically generating a basic tier of documentation for legacy software, and is also a reliable starting point for performing further, higher-level program understanding processing. Our type system defines special storage-types carrying along details on the size, in-memory representation and format of COBOL primitive data types, and a disjoint union type called flow-type that keeps track of the multiple storage-types a variable may assume when reused with different data types throughout the program - a rather common practice in COBOL. The type analyzer follows the control-flow of the program through procedure calls, branches and jumps, possibly looping, and terminates when no more changes occur in the typing context. We formalize the analysis and present our prototype implementation of the system, which includes a parser for COBOL sources (a challenging task in itself). The analyzer consists in a stand-alone application written in F# that supports several COBOL language features used in real-world programs, and statically detects a number of additional error-prone situations, such as arithmetic operation overflows and possible runtime data corruptions due to misalignment or misfit in assignments.

Similar principles may successfully be applied within an apparently distant context: for validating inter-component communication of Android applications. In the second part of this thesis we propose another type analysis technique for reconstructing the types of data within Intents - the building blocks of message passing in Android. As with COBOL, we present our implementation of the analyzer which consists in an ADT (Android Development Tools) Lint plug-in written in F# and Java, which performs a number for static checks on Android code that the Java compiler cannot perform due to the design of the Android API. Being ADT Lint fully integrated within the familiar Eclipse development environment, our analyzer effectively provides a a second-tier Java type checker for Android programs.

Acknowledgments

Several people contributed to the work presented here, either consciously or unconsciously. My special thanks go to my parents Pulcherio and Fiorella and my sister Allegra for their patience and support. To my supervisor Prof. Michele Bugliesi for his experience and for the freedom and trust he has put in me. To Prof. Cortesi for the nice discussions over the principles of static analysis. To my friends and colleagues Stefano Calzavara, Alberto Carraro and Giuseppe Maggiore for the interesting and long debates on type systems and language design. To Luca Zorzi, Francesco Restifo, Tobia Zambon and Alessandro Frazza for having worked on some aspects of this project over the years.
And of course to Aliel, for everything else.

Contents

Preface vii

1 Introduction 1

1.0.1 Plan of the thesis . . . 4

2 Parsing COBOL 5

2.1 Introduction . . . 5
2.2 Overview . . . 5
2.3 Parsing Algorithms . . . 6
2.3.1 Further parsing challenges . . . 9
2.4 Parser Generators . . . 10
2.4.1 YACC - Yet Another Compiler Compiler . . . 10
2.4.2 ANTLR - Another Tool for Language Recognition . . . 11
2.4.3 Basil . . . 11
2.4.4 Yapps - Yet Another Python Parser System . . . 11
2.4.5 Apaged - Attributed Parser Generator for the D programming language . . . 12
2.5 Parsing COBOL . . . 12
2.5.1 Challenges for COBOL parsers . . . 13
2.5.2 Available COBOL Grammars . . . 13
2.6 Our Parsing System . . . 14
2.6.1 Implementation considerations . . . 14
2.6.2 The LALR approach . . . 16
2.6.3 Framework structure . . . 17
2.7 Parsing misc divisions . . . 18
2.8 Parsing the Procedure Division . . . 19
2.8.1 Emulating Island Parsing . . . 20
2.8.2 Errory Recovery and Unknown Constructs . . . 23
2.8.3 Error reporting . . . 28
2.9 Experimental Results . . . 28
2.9.1 Project goals . . . 31
2.9.2 Case study considerations . . . 33
2.10 Future work . . . 33

3 Typing COBOL 35

3.1 Introduction . . . 35
3.1.1 Overview . . . 37

3.1.2 Comparisons and Motivation . . . 39
3.1.3 Idealized Language Syntax . . . 40
3.1.4 Storage Types and Flow-Types . . . 40
3.1.5 Environments . . . 44
3.1.6 Coercion of L-Values . . . 44
3.1.7 Loops and Convergence . . . 45
3.1.8 Ambiguity . . . 46
3.1.9 Definitions . . . 47
3.1.10 Properties of the System . . . 49
3.1.11 Type Rules . . . 51
3.2 Experimental Results . . . 52
3.3 Future Work . . . 58

4 Typing Android 59
4.1 Introduction . . . 59
4.2 Motivation . . . 61
4.2.1 The problem: Untyped Intents . . . 61
4.2.2 The Proposal: Inferring Intent Types . . . 64
4.2.3 Challenges . . . 65
4.3 Type System . . . 68
4.3.1 Formalization . . . 71
4.3.2 Environments and Judices . . . 74
4.3.3 Partial Evaluation Rules . . . 75
4.3.4 Type Reconstruction Rules . . . 81
4.4 Post-Typing Checks . . . 87
4.5 Implementation: `Lintent` . . . 88
4.5.1 Engine Overview . . . 89
4.5.2 Limitations and extensions . . . 91
4.6 Implementation Highlights . . . 91
4.6.1 Support for Java Annotations . . . 92
4.6.2 Monads and CPS . . . 93
4.6.3 Active Patterns as API Detectors . . . 96
4.7 Experimental Results . . . 97
4.8 Future Work . . . 101

Bibliography 103

List of Figures

2.1 Overall prototype architecture 17
2.2 An IF-ELSE grammar for solving the dangling-else problem 31

4.1 `Lintent` architecture 88

List of Tables

3.1 IL Syntax.. 41
3.2 Type Rules for Programs and Body. 51
3.3 Type Rules for Statements. 53
3.4 Type Rules for Arguments. 54
3.5 Type Rules for L-Values. 54
3.6 Type Rules for Literals. 55
3.7 Type Rules for Expressions. 56

4.1 Abstract Syntax of Java Types, Expressions and Statements.. 69
4.2 Syntax of Types for Intent and Components.. 70
4.3 Partial Evaluation Rules for Java Expressions.. 76
4.4 Partial Evaluation Rule Samples for Java Operator Expressions and Literals.. 77
4.5 Partial Evaluation Rules for Java Statements.. 79
4.6 Type Reconstruction Rules for Java Statements.. 84
4.7 Type Reconstruction Rules for Java Expressions.. 86

Preface

This thesis is the result of nearly 4 years of research over two main topics: analyzing COBOL legacy code and, from late 2011 on, statically checking Android programs. The material here presented is sometimes novel, sometimes an updated and reworked form of material written or published over the last years. In particular:

- a small portion of the COBOL parsing system has been co-implemented by Francesco Restifo in 2010, a Master student back then, and the whole topic has been the subject of his Master thesis;
- the COBOL typing system has been published here [66] and then selected for inclusion in [67]; COBOL analysis as a wider topic has been subject of research by two more Master students throughout 2009, 2010 and 2011: Luca Zorzi and Tobia Zambon, who respectively designed a prototype of a UI for the COBOL analyzer and a basic system for data-flow analysis built on-top of the type-flow model;
- the Android type reconstruction system is novel material, though a security validation subsystem (which this thesis will not deal with) and the analyzer implementation are currently presented here [11] and hopefully going to be published soon; the Java front-end of the ADT Lint plug-in component within `Lintent` plus an upcoming system for information-flow is currently being the subject of another Master thesis by Alessandro Frazza.

Two distinct prototypes have been in development over the years and are discussed on this thesis along with the formal models they implement. They are both available for download and examination on GitHub:

- the `COBOL Analyzer` at `https://github.com/alvisespano/CobolAnalyzer`;
- and `Lintent` is an ADT Lint plug-in at `https://github.com/alvisespano/Lintent`.

1 Introduction

The Context. In the history of programming languages probably no one has lived as long as COBOL. Designed in the late 1950s, it has been used along 50 years for writing approximatively 75% of business application, according to Datamonitor analysts [7]. The reason of such a success is manifold: COBOL similarity with plain English, its self-documenting and human-readable syntax, together with a strong support for common business processes and data structures have all been key elements of diffusion like batch operations on records. As the worldwide IT panorama evolves, millions of programs written in COBOL have to be maintained and adapted to new needs and hardware infrastructures. The pensioning of old mainframes only implies that hardware is replaced by newer and faster systems, as the New York Stock Exchange Group, Inc proved in 2006, when it migrated all its 800 COBOL programs on a modern server cluster. The real problem however, remains maintaining the enormous amount of existing sources as their creators are slowly retiring [24], taking with them the companies' only source of thorough knowledge about those programs. In order to balance this loss, interest is growing towards automated COBOL analysis and documentation tools.

COBOL similarity to spoken English, besides having probably contributed to its early success, has been welcomed as a true novelty by the language scene back then, though that has been the primary source of challenge in COBOL code analysis ever since: starting with the parsing phase, throughout advanced data-flow analysis techniques either via abstract interpretation or other approaches, COBOL variety of constructs represent a huge obstacle for any reasonably in-depth Program Understanding approach.

Let's now freeze COBOL for a moment and inspect a diametrically different world. Mobile phones have quickly evolved, over the past few years, from simple devices intended for phone calls and text messaging, to powerful handheld PDAs, hosting sophisticated applications that perform complex operations, manage personal data, offer highly customized services: in poor words, the complexity of mobile software has quickly converged to that of desktop software. This evolution has attracted the interest of a growing community of researchers, though mainly on mobile phone security and on Android security in particular. Fundamental weaknesses

and subtle design flaws of the Android architecture have been identified, studied and fixed. Originated with the seminal work in [30], a series of papers have developed techniques to ensure various system-level information-flow properties, by means of data-flow analysis [35], runtime detection mechanisms [28] and changes to the operating system [33]. Somewhat surprisingly, typing techniques have instead received very limited attention, with few notable exceptions to date ([13], and more recently [9]). And anyway all these are subjugated to security and permission analysis: no one seems to take care of how code is written, how much the host language (Java) is exploited for statically validating application, besides security or other non-purely-linguistical considerations. In our opinion, the potential extent and scope of type-based analysis has been so far left largely unexplored.

The Problem. Programming language designers have learnt a number of important lessons in the last three decades, arguably one of the most valuable of which is that validating code by means of a strong static type system leads to more robust and reliable programs. Though not all languages or development platforms, of the past as well as modern, seem to take fully advantage of this principle: COBOL and Android are two examples. COBOL is a language representing the past of computer science, tied to the world of large business applications and industry, while Android represents the present and the future of computing, tied to the world of mobile technologies, apps, next-generation devices and open-source development. Despite their distance in time, both share a wide commercial success as well as the same design flaw: they impose little to no type discipline to developers.

As said above, COBOL has been extensively used over 30 years by the business world and is still used by banks, insurance companies and finance nowadays. Several systems written in COBOL are still up and serving today and the industry is facing a novel problem in the history of computing: dealing with millions of lines of code belonging to big legacy software. Android, on the other end, is developing wider and faster than possibly any other framework in the history of open-source technologies - mobile and not -, it is used by millions of users worldwide and is facing a different novel problem: dealing with millions of line of code written by thousands of non-expert.

COBOL and Android do share something then: they have problems related to either the understanding or validation of source code, and they both need something for making applications either more robust or readable. The reasons behind these issues have roots in recent or past history - and a common solution, as we'll see below. COBOL data types come from an era when memory was a valuable resource and had to be allocated and handled by the programmer: variable reuse was an everyday practice, back then, and any kind of type discipline was rather seen as an obstacle than a tool for enforcing the program safety. Android, instead, is a modern and freshly-designed platform based on Java 5 as host language but it did not take advantage of all the latest achievements in the field of API and library design, enforc-

ing a programming pattern heavily based on inter-component communication which relies on language mechanisms that are basically untyped, or unchecked statically in general.

Moreover, in the COBOL world the maintenance of existing applications, still running and serving, is an open issue for many companies that have not yet undertaken the crucial decision of migrating to a more modern platform. And even those who did, most likely had to deal with a major challenge: understanding what those million lines of code do and what business processes they were originally meant to implement. Symmetrically, in the Android world developers every day incur into tricky bugs related to the lack of compile-time validation of the code by the Java compiler, due to the loosely-typed approach adopted by the API. Being Android a new platform, there's no literature on its history yet (except for a quick overview on Wikipedia [3]) but we argue that the reason why the Android API is so naive type-wise comes from the speed at which Android has been spread as an open-source development platform. Trivial choices made by the designers for core mechanisms, such as Intent content being untyped, might have been revamped if Android had not spread

The Proposal. Applying existing Program Understanding techniques to COBOL could be a way for aiding IT specialists in charge of a porting - but useful low-level information must be extracted from the source code in order to get any higher lever technique yield to meaningful results.

In the first chapter of this thesis we delve into COBOL: we propose a LALR parser which builds a representation of COBOL by means of a simpler and cleaner idealized language along with a special error-recovery mechanism that makes the parser tolerant to unknown constructs and statements, simulating Island Parsing techniques[1]. In the second chapter we introduce a calculus along with its typing rules defined on top of this idealized language: the type system defines special *flow types* for tracking the multiple types a variable may assume at any given program point. We believe that the multiple types of variables reused in programs are a way for both validating and understanding COBOL programs, that's why our approach is based on type reconstruction as a form of static analysis. Our system is capable of reconstructing the type-flow of a program throughout branches, jumps and loops in finite time by tracking type information on reused variables occurring in the code until no more changes occur in the typing context.

We implemented a COBOL analyzer implementing both the parser and the type system, which is capable of detecting a number of additional error-prone situations, type mismatches and data corruptions due to misalignment or misfit in variable reassignments involving datatypes having incompatible in-memory representations.

Switching to Android, our approach is the same: the core goal is not program

[1]COBOL is a complex language with a great amount of syntactic constructs supported by different specifications and revisions - supporting them all would be a cumbersome task.

understanding now, but type analysis is a great tool for validating code. We make a step towards filling the gap aforementioned: checking Android apps by analyzing type from the source code. We developed a calculus to reason on the Android inter-component communication API, and a typing system to statically analyze and verify the interaction of intent-based communication among components.

Based on our formal framework, we then develop a prototype implementation of `Lintent`, a type-based analyzer integrated with the Android Development Tools suite (ADT). `Lintent` offers a number of static checks, the core of which is a full-fledged static analysis framework that includes intent type reconstruction, manifest permission analysis, and a suite of other actions directed towards assisting the programmer in writing more robust and reliable applications. Enhancing the Android development process is increasingly being recognized as an urgent need [14, 32, 29, 50, 27]: `Lintent` represents a first step in that direction.

1.0.1 Plan of the thesis

This thesis is divided into two macroscopic topics: COBOL and Android. Chapter 2 and 3 are about COBOL and inspect, respectively, the problem of parsing and typing it; chapter 4 switches to Android typing. Every chapter first deals with the formal aspects of the problem, and then delves into implementation details and shows some experimental results.

Chapter 2 is the less formal and more practical, as it basically describes a parsing trick for parsing a language with no strict grammar as COBOL. Its experimental section is thus more focused on convincing the reader that the proposed mechanism does work by means of examples that test the various features of the parsing techniques.

Chapter 3 and 4, instead, offer an in-depth description of the type systems designed for COBOL and Android based on inference rules over type judices. Notably, they do not bypass details that are typically skipped in literature: one of our goals is to provide a formal model of a type system as close as possible to its implementation, which has to deal with real-world programs and has therefore to support a number of language of constructs that makes things complex.

In fact, both the COBOL and Android analysis systems proposed have an implementation and are mentioned throughout this thesis: they represent a sensible part of the research development work that has been done over the past years. The two implementations are distinct programs, both available for download in executable as well as source code format for examination.

Finally, all core chapters have their own brief future work section: being rather different topics and having already spent some time investigating most of the extensions there proposed, we believed it made more sense.

2
Parsing COBOL

2.1 Introduction

In this chapter we present an approach to COBOL parsing and develop a prototype implementation, which serves as a front-end for a type analysis system aimed at program understanding by reconstructing types. Thanks to a robust parsing strategy, the prototype is tolerant with respect to unsupported and ill-formed constructs. A mechanism has been developed to provide support for partially specified syntax, needed in COBOL because of the many dialects around and the presence of embedded foreign code; the same mechanism has been used to isolate fragments that are not relevant for the back-end analysis phase. The implementation of these features takes advantage of the efficiency of the LALR parsing algorithm and the reliability of Yacc, the de-facto standard parser generator. A testing phase on real-world legacy sources has confirmed flexibility to unexpected input configurations even with a partially-defined language grammar specification.

2.2 Overview

A Formal Language is a (potentially infinite) set of strings, made up of symbols from an Alphabet. Strings (or words) of a language are generated applying generative rules of a Grammar. In this work, we are interested in a particular type of grammars, Context-Free Grammars (CFGs) [6]. First formalized by Chomsky and Schützenberger in 1956 [15], these are defined as follows:

Definition. A Context-Free Grammar G is a quadruple (V, Σ, P, S), where:

- V is a finite set of nonterminal variables;
- Σ is the alphabet, i.e. a finite set of terminal symbols;
- P is the set of production rules in $V \times (V \cup \Sigma)*$
- $S \in V$ is the grammar start symbol.

As we can see from this definition, the left-hand side of a production rule consists of one non-terminal, while the right hand side may contain both terminals and nonterminals. Given a sequence of terminals and nonterminals (called *sentential form*), a *derivation step* replaces one non-terminal in the left-hand side with the corresponding right-hand side of one of the grammar's productions. *Leftmost* and *rightmost* derivations are derivation strategies in which the nonterminal to be replaced is always the leftmost or rightmost in the sentential form, respectively. The language generated by a grammar $L(G)$ is the set of strings that can be produced by a finite number of derivation steps from the start symbol; more formally, $L(G) = \{w \in \Sigma * | S \overset{*}{\Rightarrow} w\}$. A grammar is *ambiguous* if there exists a string that can be generated by two or more different leftmost or rightmost derivations.

2.3 Parsing Algorithms

The main application of Context-free grammars in Computer Science is parsing. Parsers traditionally work together with scanners, which split the input into tokens according to user-defined lexical rules. Since scanners and parsers are implemented as distinct functionalities, different levels of interaction between them are possible. The parser input consists of a sequence of tokens, that can be produced either before the parsing begins, or on demand. The *lookahead* is the number of not consumed input tokens that a parser considers when deciding which action to take in a certain state. Although the described situation is the most common, there are parsers which do not rely on tokenizers. *Scannerless* parsers allow lexical definitions to appear into the grammar. Each character of the input is treated as a terminal symbol; using regular syntax appearing in the grammar definition, tokens can be identified as parsing proceeds. Section 2.3 introduces an application of this approach in the field of island parsing, while a critical discussion on scannerless parsing can be found in [82]..

The remainder of this section briefly presents the main existing parsing strategies.

Top-down Parsing. Top-down parsing tries to find a leftmost derivation for the input, by recursively exploring the right-hand sides of grammar productions beginning from the start symbol. It is also called LL parsing, because the input is read from **L**eft to right producing a **L**eftmost derivation. Nonterminals are expanded as they are encountered, forming a tree structure whose leaves are compared with tokens: when a match is found the parser proceeds looking for further matches, descending along the right hand sides of productions.

Left recursion in grammars can lead to infinite recursion in top-down parsers, as the parser doesn't consume any tokens while going deeper into the grammar. To solve the problem, grammars can be rewritten to eliminate left recursion. Alternatively, it is possible to implement techniques that cope with the problem, recognizing this type of situations at runtime. As for other parsing techniques, ambiguity is a

concern for top-down parsers, as there is more than one alternative path to generate certain inputs.

Bottom-up Parsing. A symmetrical approach is Bottom-up parsing, also called LR parsing; it consists in reading the input from **L**eft to right, producing a **R**ightmost derivation and building the parse tree in a bottom-up fashon, starting from the leaves. At each step, a bottom-up parser compares tokens with right-hand sides of grammar productions: once a match is found, the right-hand side is replaced with the corresponding nonterminal (this is called a *reduce* action). The process continues, *shifting* input tokens in search for another match. Parsing successfully ends when the start symbol is reduced. Situations can occur in which both a shift and a reduce action, or alternatively two different reduce actions, may be legitimate. The former is called a *shift/reduce* conflict, and is usually resolved by choosing shift as the default action, as in most cases that corresponds to the intention of the grammar developer. The latter occurs when more than one production rule could apply to a certain sequence of tokens: such conflicts usually indicate the presence of an error in the grammar definition. Ambiguity is cause of concern for bottom-up parsers as well; some generator tools allow the user to assign priorities to productions, in order to give precedence to a particular derivation path among those that possibly generated the input. LALR parsers (LookAhead LR parsers) use lookahead tokens as additional information for making decisions in ambiguous situations.

Generalized Parsing. While top-down and bottom-up strategies both accept a subset of context-free grammars (typically LL(1) for top-down parsers, LR(1) for bottom-up parsers), generalized parsing algorithms allow the use of arbitrary CF grammars. Most of the available algorithms deal with conflicting parsing states using some form of *backtracking*: when encountering a conflict, this strategy consists in repeatedly choosing among the possible actions, undoing parsing if the current path leads to a parse error, until the input is successfully consumed [1]. The available implementations mainly differ in the strategy used to explore the search space, which consists of all possible parses of the input. These choices also affect the complexity of each algorithm, as in worst case scenarios parse times can grow exponentially. Ambiguities are addressed manually specifying the order in which to try ambiguous alternative productions. For a complete discussion of generalized parsing refer to [72].

Island Parsing. Describing a language in full with a grammar can sometimes be hard and time consuming; it can even be unnecessary in cases when only certain constructs of the considered language are interesting, and the remaining might safely

[1] Backtracking can also be integrated into top-down and bottom-up algorithms; however, this technique is not always useful in practice when dealing with LL(1) or LR(1) languages.

be ignored. *Island Grammars*, introduced by Moonen [53], provide detailed productions only for specific parts of the language, called *islands*, leaving the remaining parts, the *water*, as undetailed as possible. More precisely, the set of productions P of an Island Grammar can be thought as the union of the following three sets[2]:

- I, the set of detailed (island) productions, corresponding to the constructs of interest;
- W, the set of liberal (water) productions, used to match uninteresting parts of the language;
- F, the set of *Framework* productions that define the overall structure of the language and connect islands with water.

Other definitions have been used referring to island grammars in literature. When water productions appear within a fairly complete definition of a language, we are in presence of a so-called *Lake Grammar* (a name which underlines the limited extent of uninteresting syntax); on the other hand, when islands appear inside a foreign language (e.g. in the case of an embedded language), the expression *lakes with islands* is used.

A problem with island grammars is that the border between islands and water can sometimes be unclear, and the parser might not behave as expected. *False positives* are uninteresting syntactical structures that are accepted by the grammar as islands; in the opposite situation, *false negatives* are interesting parts treated as water.

Island Parsing for source analysis

We refer to Island Parsing to indicate parsing solutions aimed at implementing the behavior of an island grammar. This approach has been used in difficult parsing scenarios, where the ambiguity and/or context-dependentness of the input, the presence of embedded sublanguages or syntactic irregularities discourage writing a complete grammar specification. It has also be used as the preliminary step for automatic documentation extraction from COBOL legacy systems, which is a field of active research; recent literature on this topic is presented in the reminder of this section.

SDF [10, 37] is a modular, context-free syntax definition language. It is part of the ASF+SDF Meta-Environment [10], an incteractive language analysis environment that includes an implementation of a scannerless generalized LR parser generator (SGLR) [74]. Since the formalization of island grammars [55], SDF has been often chosen as a generator for island parsers. First attempts in this area are sketched in [76], although scarcely detailed. An extensive discussion about the expressive power of SDF in relation to island grammars can be found in [64].

[2]For a more formal definition, see [53].

A slightly different approach is described in [45], where the concept of *skeleton grammar* is introduced. Since the behavior of an island grammar in terms of false positives and negatives can be hard to establish, a method for controlling this aspect is described, assuming a fairly complete grammar of the language (which is called baseline grammar) is available. Needless to say, this is rarely the case, the point of island grammars being *not* having to specify a complete syntax specification. However this approach leads to interesting results, as the behavior of the baseline grammar with respect to the constructs of interest can be preserved in the island grammar.

Island parsing leads to the desired results when focusing on a specific subset of language constructs, but a full coverage of all COBOL real-world input configurations is still a difficult task. The point is made particularly clear in [40], where the entire process of creating a grammar specification for the language is described.

Another interesting result doesn't directly involve COBOL, but is closely related to it. *TXL [16]* is a special-purpose language designed for grammar-based language manipulation. It allows the extension and modification of existing grammars, as well as a powerful term rewriting mechanism. It has been used in [71] to develop an island grammar capable of extracting constructs of a particular language from a multilingual input (their case study involved *asp* web pages, containing (malformed) HTML, Visual Basic and JavaScript fragments appearing in a nested and interleaved fashon). The same approach can be applied to legacy COBOL, which allows embedded SQL or CICS fragments to appear.

2.3.1 Further parsing challenges

Ambiguities and context-dependent lexical analysis are a common real-world scenario when parsing computer languages (as a modern example, the C++ specification [5] describes the ambiguities in the language). To illustrate this, consider user-defined typenames. A scanner must be able to identify the syntactic category of names (eg. variable vs. type identifiers), even if they match the same regular expression. To tackle these issues, context information is usually stored at parse time and made available to the lexer by user-defined data structures, for example an early symbol table. The practice of sharing state and context information is referred to as *lexical feedback*, while a scanner that uses contextual information is said to be *context-aware*.

Another issue that parser generators need to addess is error handling. A parse error occurs if the input text doesn't comply to the grammar specification. Parsers implement very different behaviours in this regard. The algorithm may abruptly stop, reporting the error token and some state information. Error recovery mechanisms can try to resume from the error skipping some tokens until a point where normal operation can be resumed (a *resynchronization point*), inserting some tokens to match the current grammar production, or both. A complementary approach is to

allow the user to integrate some error-related information into the grammar, usually in the form of an error production (see below).

The next section provides an overview of some interesting approaches to the problems mentioned above, implemented by different parser generators. Particular emphasis is put on Yacc, the tool which has been chosen in this work.

2.4 Parser Generators

2.4.1 YACC - Yet Another Compiler Compiler

YACC [41] is a LALR parser generator, which traditionally works in combination with the Lex lexical analyzer [48]. Lex+Yacc is the de facto standard lexer-parser combintion (it has even been named *ubiquitous* [64]). The scanner functions as a subroutine of the parser, and is invoked on-demand when a new token is needed. The interface between lexer and parser can be manually expanded, making it easy to add custom code, eg. for the purpose of lexical feedback. Many implementations of Lex and Yacc are available; originally written in C, portings for a variety of languages are available nowadays (OCaml [59], F# [70], Java [65] and C# [42] among the others).

Yacc allows the user to specify error productions using the `error` keyword on the right-hand side of a production. In case of a parse error, Yacc will skip input until it finds three tokens that might legally follow the non-terminal containing the error production. Recovery is then successful and parsing continues (this is called the *three-token-rule*). If a false start is made, meaning that parsing did not resume at the correct grammar position, parsing is aborted, reporting an error. The number 3 was chosen arbitrarily; it appears reasonable, because a lower value might not be sufficiently restrictive and increase the probability of a false start, while a higher value might represent a too high requirement to be met in practice. In spite of that, Yacc's recovery behavior is sometimes hard to predict.

To help control the recovery process, the user can specify an additional token after the error keyword, and in this case Yacc will skip ahead until this token appears in the input, making it possible for the user to manually specify a resynchronization point. If, however, a second error is detected within the following three tokens, Yacc skips ahead without warning, using the three-token-rule to detect a new synchronization point.

Error handling behavior can also be determined manually. For that purpose, Yacc exposes utility functions like `yyerrok()` and `yyclearin()`. The former forces Yacc to exit error mode, the latter clears the lookahead token. It is so possible to skip the token that caused the error, manually seek to a synchronization point in the input (i.e. repeatedly invoke the lexer), and resume parsing. Unfortunately, these utilities are not available to the programmar in the functional implementations (Ocaml and F#).

2.4.2 ANTLR - Another Tool for Language Recognition

ANTLR [61] is a backtracking recursive-descent parser generator which comes with a lexer implementation. The lexical specification for tokens and the grammar are defined in the same file. Tokens can be declared at the beginning of the specification, as well as directly referenced in the production rules. Before parsing, ANTLR tokenizes the entire input, then feeds the parser with tokens from an internal data structure. This makes it impossible for the programmer to influence the lexing phase using contextual information. Although some independent users have been able to re-implement the lexer-parser interaction for on-demand scanning, this is not the default behavior, and it is not officially supported.

The last major release v.3 introduced the ability of automatically adjusting the lookahead size for every grammar production[3]; previous versions required the user to manually specify it in the grammar file.

ANTLR allows semantic actions to appear at any point of a production's right hand side. Also implemented in Bison [34], this feature can be useful when triggering semantic actions for context-aware lexing, because it allows a more natural way of writing the grammar (this practice is referred to as "Lexical Tie-ins" in Bison).

Parse errors are raised by means of an exception mechanism. Semantic actions can contain user-defined code to catch them: this way, meaningful error reporting can be implemented, and errors can be manually handled. Some utility functions are provided as well: `LATEXT(n)` returns the n-th lookahead token, while `zzconsumeUntiltoken(tok)` causes ANTLR to skip all tokens until `tok` appears in the input.

2.4.3 Basil

Basil is a LR(1) parser generator which supports context-aware scanning. This is achieved through the use of an integer attribute which is automatically passed to the lexer when requesting a token. This number, called *lexical state*, represents the position of the parser in the current right hand side, and can be used for disambiguation purposes. Regarding error recovery, Basil allows trying any number of custom recovery strategies in case of an error, and provides functionality to skip or add tokens to the input stream for this purpose.

2.4.4 Yapps - Yet Another Python Parser System

Yapps is al LL(1) parser generator written in Python. It is interesting for our purposes because of its context-sensitive lexing strategy. At each step, when requesting the next token, the parser passes the scanner a list of valid tokens, actively contributing to the resolution of lexical ambiguities.

[3] the class of grammars which it can accept is thus called LL(*), where the '*' indicates that an arbitrary finite lookahead is supported.

2.4.5 Apaged - Attributed Parser Generator for the D programming language

Apaged, a backtracking generalized LR parse generator, supports context-aware scanning through custom code blocks that can be integrated into the lexing routine to manually control its behavior. It features a distinctive error handling mechanism. It allows the user to manually specify the location of synchronization points via a special keyword, which can appear in any position in a right hand side. In case of an error, this information is used to locate the exact grammar location where parsing can be resumed. As an example, statemets are usually separated by a semicolon or a similar token. If a synchronization point is set after it, the parser will ignore tokens of an eventually ill-formed statement[4], avoiding the loss of the already parsed ones and reverting to normal behavior for the next statement.

Semantic actions are only executed after the input has been completely parsed. As the user is able to manually invoke semantic actions associated with each nonterminal, it can execute them an arbitrary number of times, implementing multipass analysis if desired. However, this also means that the user cannot influence the construction of the parse tree.

2.5 Parsing COBOL

Unlike modern programming languages, a COBOL program has to obey certain formatting rules, a heritage that comes from its punch-card history. The first six characters of each line are reserved for line sequence numbers, the seventh for the continuation character[5] and the comment marker, and positions 72 to 80 are reserved. Characters 8 through 11 are known as Area A and are used mainly for division, section and paragraph header names and for level indicators (2-digit integers needed to associate fields to records). Area B (characters 12 to 72) contains the actual code.

A COBOL program is structured hierarchically in four Divisions (Identification, Environment, Data and Procedure Division), logically separated containers for file attributes, I/O options, data declarations and statements. Each of these contains various subdivisions or Sections, which in turn consist of labeled Paragraphs, named groups of colon-separated Sentences which can comprise multiple Statements.

A specification for a COBOL parser should have a precise knowledge of the syntactic details for each hierarchical level. Some formatting constrains require special attention. For example, line numbers can appear anywhere a line break can

[4]The error recovery routine discards the error token and all symbols encountered after passing the marked position.

[5]The continuation character allows using the following line for constructs that do not fit into a single line, e.g. long string literals.

occur, a sure cause of grammar explosion. For that reason a pre-processing phase is often employed to remove these semantically irrelevant elements.

2.5.1 Challenges for COBOL parsers

First of all, a number of vendor-dependent language implementations are available, each extending the standard syntax with specific constructs, keywords, operators and even statements. Different compilers accept different dialects, and are differently restrictive about the formatting of sources. Although this can't be considered a blocking difficulty, it makes it impossible to precisely identify the language to be parsed in the general case, and forces one to focus on a particular vendor implementation. Moreover, the variety and freedom of the allowed syntax and the vastity of the language make it hard to even try and define a complete grammar for one precise version (the C programming language defines approximately 40 keywords, while the COBOL 85 specification mentions over 400; statement can have up to 6 syntactically correct formats, each having multiple optional parts).

The hardest challenges, however, are related to context-sensitiveness and ambiguities in the language specification. Context-sensitiveness arises mainly from tokens having more than one correct meaning in the same program, depending on the enclosing division (picture declarations are the clearest example of that) or on previous declarations (e.g. the way money amounts are defined depends on the specified currency symbol). A simple example of ambiguity is arbitrary nesting of statements having optional parts. Since COBOL does not allow bracketing and scope delimiters can be omitted, in many cases optional clauses could be associated with more than one of them [51].

Although most ambiguities can be resolved rewriting the corresponding grammar productions and using associativity rules, there are cases where ambiguities are context-dependent. Such situations cannot be solved by the parser, and may be handled performing an extra pass on the AST [75].

2.5.2 Available COBOL Grammars

Despite COBOL's unique parsing challenges, attempts to develop a COBOL grammar in BNF form (or one of its variants) for use with a parser generator have been made. Rather than writing it by hand, [47] derived a lexical and grammar definition from IBM's language specification by means of a semi-automatic process. This experimental grammar has been used as a starting point for two different projects. The first [63] is a JavaCC [52] specification that can be used to create a COBOL parser in the Java programming language. Input sources require preprocessing to comply to this grammar; a prototypical preprocessor implementation is available. The second project is the OpenCOBOL Compiler [1], an open-source compiler for

the language. This is the only project where recent activity has been made, even if it is not clear wether it still is under development (the last visible modification dating early 2009). As of today, no grammar-based parser is a fully working real-life solution for legacy COBOL code.

2.6 Our Parsing System

The above discussion should not discourage alternative approaches to COBOL parsing. As we will see in section 2.3, restricting the focus on a subset of the language special purpose parsers can be generated in an automated way.

In our work, we target COBOL-85 [4] code running on the IBM z/Os platform. A COBOL parser is needed as the front-end of a type- and data-flow analysis software system. For this purpose, a fully-featured language support would be a waste of effort, while the interesting subset of interesting constructs can be easily defined. For example, constructs that do not affect the type of stored data are not significant; compiler directives, database access statements and I/O primitives, among others, also fall in this class, simplifying our task in a way that can be handled using standard parser generators. On the other hand, since such constructs may still reveal useful information for the analysis stage, they are not to be ignored; rather they represent input fragments having a lower relevance level, and need to be isolated from interesting parts.

Due to partial language coverage, inputs outside the systems' syntactical scope can appear in source files: a *robust* parsing strategy is needed to cope with these situations. Moreover, given that unknown or unsupported constructs may be useful for documenting purposes, they cannot be simply skipped, even when their syntax is not known.

The goals of this work can be summarized as follows:

- parsing a task-focused subset of COBOL using a widespread automated tool;
- accepting portions of input having a lower level of relevance, not having to specify their full syntax;
- implementing a robust strategy towards unknown/unsupported parts of input, avoiding parse errors, and keeping track of the symbols encountered.

2.6.1 Implementation considerations

After considering the many parser generators available (refer to section 2.3.1), we decided to rely on Yacc as our parsing tool. We briefly motivate our choice against the alternatives below.

Scannerless parsing. The unification of the lexical and syntactical language definitions in a single specification (along with the removal of the interface between them) is the most relevant advantage of the scannerless parsing paradigm[6]. This can certainly be positively considered for small projects, but it is a certain source of confusion and poor maintainability when dealing with a language such as COBOL: the result would probably be a huge, monolythic, hardly readable specification file. This makes using scannerless parsers impracticable for our purposes.

It must be admitted though, that some other features are certainly interesting. One of these is grammar-guided tokenization: a token will not be considered if it cannot appear in a certain position.

Handling semantic actions. Another aspect of parser specification is the way semantic actions are handled. While they can be executed during parse time, *purely declarative* parser generators have a different approach. They separate the syntactical analysis phase (which yields an automatically-generated AST representation of the input) from the user code, which is executed only once the entire input has been seen. This is only applicable if a generic tree representation is sufficient, not in situations where full control on its construction is required (which is the case of this work, as section 2.6.2 explains).

The use of backtracking strategies influences semantic actions as well. Each time the parser encounters an error, it may attempt to match the input taking a different path in grammar productions; actions that were executed after a wrong choice need to be undone for obvious consistency reasons. This is only possible if they do not maintain or modify global data structures, or if they do not affect parse tree construction. Unfortunately, this is not our case. To solve these problems, [73] proposes an extension to current backtracking strategies introducing semantic actions that can be undone.

Island grammars. The idea of Island Grammar has a close affinity with the objectives of this work. The need for support of unspecified syntax is in close relationship with the idea of water. The desired support for partially relevant fragments recalls the generality in language specification which was the motivating point for introducing island grammars. The most reliable impelmentation of a parser generator capable of island parsing (SDF), unfortunately uses a scannerless approach. As described in the next chapter, a different way towards similar results has been successful using the LALR parsing technique.

[6]The ANTLR parser generator also unifies the two definitions, even though it is able to derive a lexer implementation from it

2.6.2 The LALR approach

Based on the above motivations, we decided not to rely on scannerless, purely declarative and island parser generators. Further considerations led us to the conclusion that the traditional Lex-Yacc combination would be the best choice for achieving project goals. These are summarized in this section, along with some of the advantages of this approach.

Control on AST construction. As already mentioned, the implemented parsing framework acts as the front-end of a source analysis tool. The main idea behind the development of the parser is the translation of interesting parts of COBOL programs in a well-formed, succint intermediate representation. This representation is named IL, which stands for *Idealized Language*[7], and can be defined as an imperative, typed language, specifically designed to be a suitable target of a program translation process (its syntax is described in 3.1.3). To sum up, our task is translating COBOL into an equivalent IL syntax tree built in a completely controlled manner, rather than with the help of automated tree construction utilities.

Performance. Efficiency should be kept into consideration as well. Currently stable and supported implementations of both scannerless parsers and backtracking strategies have not entirely solved efficiency problems due to the computational overheads introduced by these techniques (respectively caused by scannerless char-by-char approach and backtracking worst-case complexity). In contrast, the high performance of the bottom-up approach is vital when dealing with millions of lines of legacy code.

Integration. By the time work on the front-end of the analysis software started, the code analyzer was already under heavy development, the chosen programming language being F# [70]. Having a native, official F# implementation eliminates the problem of importing and interpreting data structures (AST) created by an external parsing tool. One more technological advantage of this choice is that a functional tool is most suitable in a setting where data structures are dynamically built in a bottom-up fashion (e.g. the single elements of a tuple have to be created before the whole touple). An imperative environment instead, forces creating data structures before populating them: this mirrors a top-down behavior. Furthermore, the functional paradigm was chosen as the programming paradigm for the analyzer, among other reasons, for its intrinsic robustness, expressiveness, maintainability, and convenience for the task of source analysis. The parser should be consistent with this choice, and share the benefits deriving from it.

Diffusion. As stated at the outset, one of the key requirements for our project was the development of a reliable and robust parsing strategy for a focused selection

[7]Refer to section 3.1.3 for the full syntax.

of the COBOL language. The de facto standard, as well as the most widespread solution for parsing, is the Lex-Yacc combination. The computational performance, the large number of target languages that are supported, including functional languages, and its many applications, make YACC the most widespread tool for syntactic analysis. Although more expressive strategies have been developed in recent years, one of the aims of this work is to show how to deal with potentially problematic inputs using this tool, bringing the advantages of island parsing into the LALR world.

2.6.3 Framework structure

Our prototype consists of three lexical and syntactical specifications for different parts of a Cobol program and an F# software library. In the general case a preprocessor might also be needed, depending on the considered input format. In our case the mainframe sources at hand do not need preprocessing, as they do not include occurrences of sequence numbers, continuation markers or characters after position 72, leaving us with an intact Area A (comments may still be present). The general architecture of the system is shown in Figure 2.1.

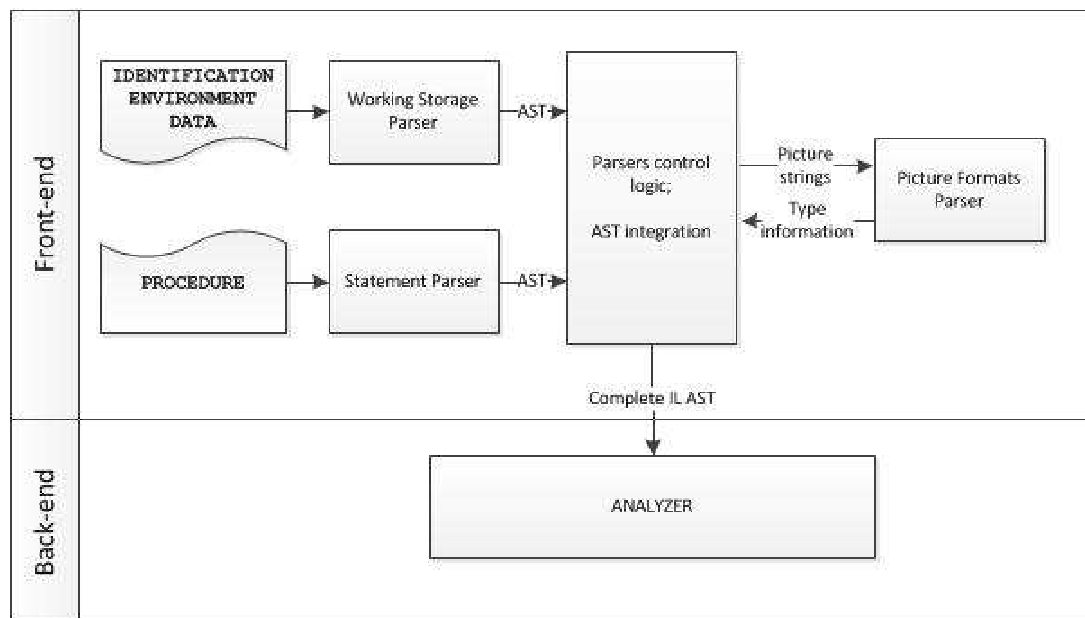

Figure 2.1: Overall prototype architecture

Considering the vastity of the language, to ease maintainability Cobol sources are conceptually organized in two logically separated macro-areas: Identification, Environment and Data divisions on one side, Procedure division on the other. A distinct lexical and syntactical specification has been developed for each of the macro areas, allowing a meaningful separation between the many Cobol configuration options and program attributes and the actual imperative code. To further simplify

the synthesis of the grammar for the Procedure division, the decision was made to focus on the smallest independent unit of code, a single sentence or statement[8] (this parser has been consequently named Statement Parser).

It should be clear that the parsing functions need to be manually invoked (once on the first three divisions, and repeatedly on each statement or sentence in the Procedure division): a software layer must be written above the parsers, to keep track of the current position on the input. Moreover, the AST construction is directed using semantic actions, and since multiple parsers exist, a syntax forest will be produced that needs to be unified in a single tree; this is another task left to this software layer.

As discussed in section 2.5.1, a further pass on selected syntax may be required to handle particularly problematic Cobol artifacts. Although this planned feature has not been implemented yet (see section 2.10), post-processing is already used in presence of Picture format strings. During the first pass, Lex is able to correctly disambiguate them from identifiers, but their internal structure, revealing type details of declared data, is not preserved (they are treated as strings instead). As this kind of information is vital for the back end analyzer, a second knowledge-extraction pass is needed to reveal the type of each data item. This may either be achieved by custom code, or by a second parser exclusively targeting the sublanguage of the strings. The latter has been chosen because of the obvious advantages of automation and ease of implementation.

2.7 Parsing misc divisions

The structure of the first three divisions in a Cobol source file (which are handled by a separate parser, as seen in the previous section) are here illustrated, for the purpose of identifying meaningful information for the analysis phase and uninteresting constructs:

- Identification division: mandatory division used to define documentary information attributes (e.g. author, company, date). It is syntax checked only by Cobol compilers.

- Environment division: consists of two sections. The Configuration section describes the computers where the program was compiled and where it will be executed. The Input-output section defines files, assigns them to I/O devices (e.g. disk, tape, printer), and specifies access-related information. Both this division and its sections are considered optional for the majority of compilers.

[8]The smallest possible unit of code in Cobol consists of a sentence, i.e. one or more statements terminated by a colon. In case only one construct appears, the sentence coincides with the lone statement.

- Data division: the File section describes the physical structure of files and their contents. The Working Storage section, the only mandatory part of the Data division, is the place where data items (variables) used in the program are declared. The Linkage section regulates sharing data between programs.

Only the Working Storage section of the Data division contains semantically relevant information for type analysis (this is why this parsing step has been named Working Storage Parser in figure 2.1); all other definitions only need syntactical (grammar) support, and do not present any challenges for parsing. The generated AST will therefore hold exclusively variable, record and constant declarations.

Let us focus on parsing Cobol picture formats, which hold precious type information for each declared Cobol data unit. When the Working storage scanner encounters the PIC (or PICTURE) keyword token, it jumps into a special state. Here it is instructed with a simple lexical definition of format strings, consisting of repetitions of legal characters, taking into account the possible positions of comma, dot and brackets[9]. When a whitespace or a period (a dot followed by a newline) is seen, lexing continues from the default state. The returned token type carries the recognized lexeme, which is stored as a strings in the abstract syntac tree.

A post-processing phase is needed for type information extraction from these strings[10]. A second parser (the Picture formats parser in figure 2.1) is defined to create an internal representation of the delcared data. As each single character has a precise semantic meaning, syntactical analysis must be performed on a character basis. Each single symbol is tokenized; semantic actions build a type representation which will be made available to the back-end analyzer. The results of this process are shown in section 2.9.

2.8 Parsing the Procedure Division

As previously mentioned, only a subset of COBOL statements are semantically relevant for the source analysis phase, thus requiring a complete syntactical definition. Although the language coverage to be anchieved is significant, it is possible to write the corresponding grammar without relevant issues by hand, taking advantage of the expressiveness and intuitiveness of the LALR approach.

Besides the aspect of parsing, it is interesting to spend a few words about the construction of the respective AST branches. A single COBOL statement may carry a rich semantical content (e.g. simple arithmetic operations can allow results to be calculated and stored in various locations, with rounding options and user-programmed overflow handling logic). Recall that an IL representation of the

[9]It can be expressed as the regular expression: `['A' 'X' 'Z' 'S' '(' ')' '0'-'9' 'V']+ (['.' ','] ['A' 'X' 'Z' 'S' '(' ')' '0'-'9' 'V'])*`

[10]Note that this second pass happens only once the whole input file has been parsed.

COBOL input is needed. This could not be achieved by means of concise semantic actions only: a library of parse-time translation utilities has been written to perform the task. Constructs that are not in a one-to-one correspondence with an IL statement are referred to as *Block statements*, as a block of IL code is originated from them. When a block statement is parsed, the corresponding library function is invoked, yelding a semantically equivalent tree forest. Sometimes it is even necessary to declare temporary IL variables (e.g. implicit loop counters). These must be given a unique name and a type, as they obviously do not appear among the Data division declarations. This information has to be retained by the framework and merged with the existing data items after parsing is completed. Detailed translation examples are shown in section 2.9.2.

2.8.1 Emulating Island Parsing

A vast class of statements appearing in the input code is not relevant for the analyzer, as mentioned in section 2.6; the program type and data flow is invariant with respect to them, meaning that their presence has no effect on it; we refer to these constructs as *Invariant statements*. Given that the parser's objective in this case is their mere recognition and isolation, specifying their exact syntax would be a waste of effort. However, the fact that some useful information, mostly for documentation purposes, can be found in these fragments can not be ignored. As an example, the appearence of particular classes of tokens in them could be relevant; in this work, this is the case of COBOL keywords and identifiers. It is meaningful to also store the whole statement in which they appeared in the input, which can be treated as text.

To sum up, what is needed is a way of defining a general grammar structure capable of accepting partially defined syntax, distinguishing some desired classes of lexemes from the others, and saving the interested line(s) of code.

Implementation

A COBOL statement always begins with a reserved keyword: the membership of the construct to the invariant class can be determined by the lexer as soon as this token is seen. If an invariant statement keyword appears, Lex produces a special token, which is only used in the grammar at the beginning of a general, multiple-line catch-all production, named `invariant_stmt`, as shown below.

```
/* invariant statements */
invariant_stmt:
   invariant_kwd anything_mode_inv_lines RESYNC
                                          { clear_buffer_at_eol <- true; $1 }

 | invariant_kwd anything_mode_inv_lines END_INVARIANT
                                          { clear_buffer_at_eol <- true; $1 }
```

```
invariant_kwd:
        INVARIANT_KEYWORD                     { keyword_token_mode <- true;
                                            clear_buffer_at_eol <- false;
                                            $1 }

anything_mode_inv_lines:
    anything_inv_lines                    { keyword_token_mode <- false; $1 }

anything_inv_lines:
    anything_inv                                { $1 }
  | anything_inv newlines                       { $1 }
  | anything_inv newlines anything_inv_lines    { $1 @ $3 }

anything_inv:
    any                                 { $1 }
  | any anything_inv                    { $1 @ $2 }

anything_mode:
    anything                            { keyword_token_mode <- false; $1 }

anything:
    any                                 { $1 }
  | any anything                        { $1 @ $2 }

any:
    ID                                  { [$1] }
  | KEYWORD                             { [$1] }
  | structured_access                   { [] }
  | lit                                 { [] }
  | bool_binop                          { [] }
  | arith_binop                         { [] }
  | logic_binop                         { [] }
  | BRA                                 { [] }
  | COLON                               { [] }
```

Let us analyze the syntactical specification first; semantic actions will be treated shortly. As expressed in the top production, an invariant statement can be terminated by an optional `END_INVARIANT` token[11], produced for all legal terminators of invariant statements; another legal terminator is the `RESYNC` token, which is a synchronization token whose role is explained in detail in the next section. An invariant construct can spread along multiple lines, each consisting of occurrences of a set of legal tokens (`any` and `anything` nonterminals) appearing in an arbitrary order. This general specification allows a wide range of syntactic combinations to be accepted,

[11] In COBOL the majority of scope terminators are optional.

withou worrying about their structure.

As one can deduce from the `any` production, the scanner generates a `KEYWORD` token for every keyword appearing within an invariant statement. To do so, it needs context information from the parser: the `keyword_token_mode` flag, activated only while parsing invariants, serves this purpose. A crucial necessity here is being certain that no tokens will be read after the `INVARIANT_KEYWORD` and before the flag is set (Yacc could already have consumed the lookahead token when `invariant_kwd` is reduced). However, the `INVARIANT_KEYWORD` token only appears once in the whole grammar specification, causing no ambiguity with respect to other productions. A lookahead parser only reads ahead if it is not able to make a decision; this is not the case of the `invariant_kwd` production, because of the unique distinguishing `INVARIANT_KEYWORD` token; for this reason, we can assure the correct operation of this mechanism.

To understand the meaning of the action code related to the `clear_buffer_at_eol` flag in the Yacc grammar above, consider that the input is seen by the parser as a token stream. For the purpose of saving the lines where invariant constructs appear, the respective lexemes must be concatenated: lexemes, together with tokens, need to be stored in a special data structure (named `token_buffer)` as they are produced by Lex. Intercepting tokens can be done before they are pushed to the parser, noting that at the moment of parser invocation, the scanner function is passed as a parameter. We simply have to encapsulate the tokenizing function with a wrapper, which sends every recognized token to the token buffer before returning it.

Now, clearing the buffer between statements is sufficient to guarantee that its contents pertain the currently parsed construct. As an invariant can spread across lines (e.g. in the case of embedded SQL queries), the buffer contents need to be preserved until its terminator is seen; this is signaled by setting the `clear_buffer_at_eol` flag. When the non-terminal corresponding to an entire statement is reduced (`invariant_stmt` in this case), our lexer wrapping function triggers the concatenation of lexemes in the buffer. This yields the desired result, i.e. the invariant input fragment, which can now be stored. At the end of the process, the buffer is cleared.

Scalability

In this work, only one differently-relevant syntactic class, the invariant class, is needed. However, it is worth mentioning that no limitation exists on their number, provided a dedicated token is reserved for each of them.

One important thing to notice is the ease of adding constructs to the invariant class: the respective statement keyword just needs to be added to the scanner specification in order to produce an `INVARIANT_KEYWORD` token. The only drawback of the described approach is the need of explicitly specifying tokens that are legal in the catch-all production. An island grammar specification would be a more

maintainable solution to this regard, because it would not require checking wether newly introduced tokens should be added to the `any` list.

2.8.2 Errory Recovery and Unknown Constructs

The most important objective for any COBOL parser is undoubtedly robustness, i.e. the abilty of being tolerant to unexpected inputs without breaking the parsing process. The isolation of unknown or (currently) unsupported language constructs is a fundamental feature also in an early development stage, as it avoids painful crashes on real-world code and allows testing of supported features without having to write limited test cases.

Errors can occur for two different reasons in the Procedure division:

- A totally unsupported/unknown statement is encountered. As already mentioned, a COBOL statement always starts with a dedicated keyword. Since no lexical specification exists for it, the identifier regular expression will be matched. Identifiers can only start a procedure division line in the case of paragraph headers, which are followed by a period: any other configuration leads to a syntax error.

- A bad/ unsupported format of a recognized statement is encountered, which also is a syntax error.

Both, of course, can also occur within a nested construct, whose integrity must be preserved. In whichever case, Yacc's error recovery mechanism (described in section 2.4.1) will be triggered. To avoid parse errors, error productions covering the above cases are needed. The position of the automatically detected synchronization point is unpredictable: it is therefore important to specify unequivocally where normal parsing can be resumed.

Implementation

The need of separators The hardest part of the task is specifying synchronization points, in a language where the help of delimiters cannot be assumed (in nested constructs there is no symbol separating each statement from the next one). Again, the fact that each statement is introduced by its own keyword is helpful, because the beginning of the construct following the unsupported fragment is exactly where recovery should occur. The question is how to recover *before* consuming the keyword token. It is clear that this cannot be done without adding some symbols to the input. Our strategy consists in generating a `RESYNC` synchronization token whenever Lex recognizes a statement-starting keyword, in order to explicitly mark the restore point. A particular kind of statements containing error productions has been developed; a discussion of each of the reported grammar rules follows.

```
error_statement:
  | error_id anything_mode RESYNC       { }
  | error_id anything_mode EOL          { }
  | error RESYNC                        { keyword_token_mode <- false }
  | error EOL                           { keyword_token_mode <- false }


error_id:
    error_id__                          { $1 }

error_id__:
    ID                                  { keyword_token_mode <- true; $1 }

anything_mode:
    anything                            { keyword_token_mode <- false; $1 }
```

Let us focus on the first rule of the `error_statement` nonterminal. The presence of two entities to be sent to the parser at once (`RESYNC` and the keyword itself) is in contrast with the fact that Lex has a single return value. To get around this, we declare a new token in the parser specification file (namely: `%token <token> ResyncKeyword`[12]), which is able to carry an argument of the same type all tokens belong to. When one of the considered keywords is seen, `ADD` for example, Lex produces a `ResyncKeyword ADD`. Recall that the tokenizer is encapsulated in a wrapping function. Whenever Lex returns a `ResyncKeyword`, the wrapper retrieves and stores the inner token via pattern matching, and returns `RESYNC` to the parser in its place[13]. The next scanner invocation will not execute Lex-generated routine if a pending token is found (`ADD` in our example).

At this point, `RESYNC` is our error production terminator[14]. It also needs to be added before scope terminators and dots (as the unsupported statement could be in last position within a sentence). This explains the operation of the first error production.

A consideration explains the idea behind the second rule. One may ask if allowing the parser to enter error mode could not be avoided in some situations, in order to maintain direct control of the parsing process in a higher number of situations. A positive answer can be given for all those cases in which a COBOL statement is completely unsupported by the used grammar. As its distinguishing keyword is not known, it is tokenized as an identifier: thanks to the same catch-all mechanism used for invariant statements (refer to the `anything` nonterminal, which role is explained

[12]The `%token` directive introduces a new type of Yacc token; this is defined by the `ResyncKeyword` ML data constructor, which carries an argument of type `token`. The result is a new kind of token which acts as a container.

[13]Note that `ResyncKeyword` never appears in grammar productions

[14]Referring to section 2.8.1, `RESYNC` also acts as delimiter for invariant statements.

in section 2.8.1), it is possible to handle these situations explicitly. The only difference with invariants is that here we cannot guarantee the `keyword_token_mode` flag to be set right after the ID, as the lookahead token may already have been consumed (some more leaf grammar productions starting with the ID token are likely to exist: a LALR parser needs to read ahead to disambiguate between them). For this reason, if a keyword follows the first identifier token, Yacc enters error mode, recovering when a `RESYNC` token is found. In other words, this implementation does not assure avoiding error mode in all cases in which entirely unsupported constructs are seen. However, this can be assumed in practice; it becomes clear analyzing the COBOL-85 specification [4], where the few statements possibly introduced by two keywords represent such fundamental constructs that either complete or partial (i.e. invariant) grammar support for them in a COBOL parser is taken for granted[15]. In the remote case in which some of them are still unsupported, the error mechanism takes over, assuring the desired parsing behavior.

Introducing the `EOL` token As the reader may notice, an additional `EOL` token appears in the last two productions of the `error_statement` nonterminal. An example illustrates why this is required. Imagine a first ill-formed constructs is encountered, followed by a totally unknown statement, both within a nested sentence, as shown below (assuming the `USE` statement is not supported, it is tokenized as an identifier).

ADD TO MY-VARIABLE
USE ERROR PROCEDURE ERR-PROC.

No `RESYNC` is produced when the latter is seen, making it impossible to distinguish it from the former. It is clear that the only possibility of correctly interpreting this input is introducing an end of line token, as no other symbol can act as a delimiter. The same issue would have happened if the first statement would have been another completely unknown construct: line breaks are needed in this case as well (this is implemented in the fourth production of the `error_statement` nonterminal). Such situations may appear frequantly: this solution broadens the tolerance of the parser to a larger set of syntactic irregularities.

Considerations

Implications of delimiters in grammar productions. The grammar excerpt below highlights the fact that this method requires minimal grammar reworking: the presence of the `RESYNC` and `EOL` delimiters is easily handled. They are both added in the `statement` production (while the presence of the latter is self-explanatory,

[15]The statements falling in this class are: `EXIT PROGRAM, FREE ALL, GO TO, STOP RUN`, and specific formats of the `PERFORM, BEGIN, DISPLAY, HOLD, MOVE, OPEN, SEARCH, SET` and `USE` statements. We can see that all ot them, maybe with one only exception (`USE`), belong to the very core of the language, appearing in almost every real-life source code.

the former has to be consumed when it appears between two supported statements); `RESYNC` must be considered before `DOT`s at toplevel, where also newlines need to be taken into account to support entirely blank lines. Their optional presence has also to be considered between clauses in multiple line statements (e.g. before `WHEN` clauses in an `EVALUATE` phrase), natural locations for beginning a new line[16]. Nested constructs are automatically supported, as they are recursive on the `statement` non-terminal.

```
procdiv_line:
    newlines EOF            { EndOfStream }
  | EOF                     { EndOfStream }
  | ID RESYNC DOT           { ActualLine (Label (locate parseState 1 $1)) }
  | newlines ID RESYNC DOT  { ActualLine (Label (locate parseState 2 $2)) }
  | sentence                { ActualLine (Statement $1) }

newlines:
    EOL                     { }
  | EOL newlines            { }

sentence:
    statements DOT          { block_of_statements $1 }

statements:
    statement               { [$1] }
  | statement statements    { $1 :: $2 }

statement:
    statement'              { locate parseState 1 $1 }

statement':
    newlines                  { Nop }
  | RESYNC                    { Nop }
  | simple_statement          { $1 }
  | block_statement           { $1 }
  | error_statement           { ... }
  | newlines error_statement  { ... }
```

Other considerations The presented method adds new token kinds to the input stream, with minimal impact on existing productions. Although a preprocessor could be used to this aim, inserting missing symbols with an arbitrary convention, the same results are here achieved during parsing. This is a more efficient solution,

[16]COBOL formats allow a great number of optional clauses to be specified, and experience tells that when line breaks occur, they are located between these clauses, rather than at any arbitrary position. Although this is not a syntactic rule, COBOL sources comply to it for obvious reasons; experience has validated this approach as a good working solution in a real-life environment.

since no additional pass is required; a particularly appreciated feature when dealing with COBOL source files, which can spread over several hundreds of thousands lines.

The described implementation of a robust parsing strategy cannot lead to false positives, because a complete specification of the interesting constructs is given. However, in some rare situations false negatives can occur. This is caused by an unwanted side-effect of Yacc's error handling mechanism in presence of an ill-formed fragment spreading across multiple lines, as in the example below (`BEFORE` is incorrectly spelled):

PERFORM SOMETHING WITH TEST BFORE
  UNTIL MY-VARIABLE ¿ 3
ADD 1 TO MY-VARIABLE.

Yacc is not able to deal with this kind of situations. In fact, after entering error mode on the first line, it correctly identifies the `EOL` token as point of recovery, expecting the beginnig of a valid statement. Instead, another syntax error is seen within the following three tokens. Judging to have made a false start, it jumps back into error mode, not recovering until input complies with the three token rule (see section 2.4.1). The first legal token allowed to occur after a sentence is `DOT`[17]: the `ADD` statement is therefore included in the error, leading to a false negative (recall that a false negative is a construct of interest unwantedly not recognized as such).

There are two possible causes of this behavior:

- No grammar support exists for a statement which spreads across multiple input lines, and some of its formats have two leading keywords (otherwise Yacc would not enter error mode, thanks to the `error_id` grammar productions). This situation, which is very unlikely to happen for aforementioned reasons, can be raidly solved by including the construct in the invariant class, i.e. adding its distinguishing keyword to those producting the `INVARIANT_KEYWORD` token (multiple lines invariants are correctly handled).

- Syntax errors are present in the input source. Parsing legacy COBOL programs with syntax errors does not appear reasonable, as these have been compiled and running, presumably for decades.

We can sum up the discussion by stating that, although the presence of false negatives cannot be categorically excluded, these might appear in the two cited scenarios; the former can be addressed in literally seconds during a testing phase, the latter is caused by an invalid input configuration and is highly unlikely to happen in the field of application of this work, which is the analysis of legacy COBOL.

[17]When looking for a synchronization point, Yacc does not consider eventually recursive nonterminals following the production in which the error occurred. In our case, error mode was entered in the `statement` production, which is automatically reduced. Now, the second rule of the `statements` nonterminal is recursive and will not be tried. Therefore, the first token legally following the error is a `DOT`, which also marks the end of the parsing process.

2.8.3 Error reporting

Our error handling mechanism for the procedure division is designed to tolerate arbitrary input (only at toplevel productions syntax must be respected, which basically means that non-empty inputs must be terminated by a dot). For what concerns the Working Storage parser, a meaningful error reporting mechanism has been provided, which is activated in the remote case an illegal input is able to break parsing unexpectedly (e.g. because of disregarded formatting restrictions). On a parse error, FSYacc allows the optional execution of a user-defined handler function (which is called `parse_error_rich`); a parameter is passed to it, namely a data structure containing information about the parser state. Useful fatcs are extracted and displayed from it; consider the following sample code:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. ERROR-REPORTING-EXAMPLE.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
DATA DIVISION.
WORKING STORAGE SECTION.
01 X PIC S9V9 COMP-3 VALUE 2.
```

The `WORKING-STORAGE` keyword is incorrectly split into separate words, leading to a syntax error. As it can be difficult to spot such a spelling mistake, additional information joins the position of the error, as we can see from the output below:

```
parse error: syntax error: token ¡ID WORKING¿ at line 6 column 1-8
shifts: TOKEN_WORKING_STORAGE TOKEN_FILE reduces:
states: 103 7 4 2
reducible prods: NONTERM_datadiv; NONTERM_program1; NONTERM_program;
   NONTERM__startprogram
```

The illegal lexeme is displayed along with its token class. The user is also informed about tokens allowed to appear in its place (the `WORKING-STORAGE` and the `FILE` section headers), undecidable state numbers for the encountered input configuration, and nonterminals currently appearing on the parser stack. Such precise and informative output is definitely helpful to quickly isolate the cause of unexpected behaviors.

2.9 Experimental Results

Section 2.9 is dedicated to illustrating the capabilities of the described parsing framework in relation with implementation requirements and project goals. Then a case study on an existing Cobol legacy system is presented in section 2.9.2.

Context-sensitive lexing

The meaning of tokens can be dependent on the context in which they appear. This implies the need of a context-aware lexing strategy, and of some form of communication between scanner and parser. Lexical feedback has been implemented to this aim, by means of status information that is maintained and updated using semantic actions in the grammar specification file. Basing on the information it is provided, Lex is able to decide which token to return. This mechanism has been used to provide support for partially relevant syntax, as seen in section 2.8.1; in particular, a special token is produced when a Cobol keyword appears within an invariant construct, making it possible to define a general-purpose catch-all production.

Picture format strings

Picture formats represent a similar class of context-dependencies. Symbols have an individual precise semantic meaning when appearing within them, different from the one they have in other contexts. In this case, however, the type-related knowledge they carry is vital for the back-end analyzer: a dedicated parser has been defined in order to acquire this information durin a post-processing phase. Consider the following Cobol program:

```
IDENTIFICATION DIVISION.
    PROGRAM-ID. WORKING-STORAGE-TEST.
    ENVIRONMENT DIVISION.
    CONFIGURATION SECTION.
    DATA DIVISION.
    WORKING-STORAGE SECTION.
    01 X PIC S9V9 COMP-3 VALUE 2.
    01 R.
        05 R-A PIC 9(4) VALUE ZEROS.
        05 FILLER PIC XX VALUE SPACES.
        05 R-B PIC 99V9.
        05 FILLER PIC XX.
        05 R-C.
            10 R-C-1 PIC 9(5) VALUE 100.
            10 FILLER PIC Z09(5).
            10 R-C-2 PIC S9(4) VALUE -20 USAGE COMP.
    PROCEDURE DIVISION.
    ADD -10 TO X GIVING R-B R-C-2.
    MOVE R-A TO R-C-1.
    STOP RUN.
```

Most of the allowed picture formats appear in this fragment: type-specific characters, sign and comma placeholders and symbol repetitions are used within nested record declarations. The results of the second parsing stage are integrated in the syntax tree built by the Working Storage parser. The process yields the following output, which is generated by an AST pretty-printing routine (being the result of a

translation process, the syntax tree is conveniently displayable as an IL program):

```
{
  R.R-B := X + -10;
  R.R-C.R-C-2 := X + -10;
  R.R-C.R-C-1 := R.R-A;
  return;
}
where
  X : bcd[S1.1] := 2
  R : { R-A : num[4] := 0,
      FILLER : alphanum[2] :=  ,
      R-B : num[2.1],
      FILLER : alphanum[2],
      R-C : { R-C-1 : num[5] := 100,
      FILLER : num[7],
      R-C-2 : bcd[S4] := -20 }
     }
```

IL type information, reported in the `where` clause appearing after the generated IL statement block, is expressed by means of a concise syntax that reflects the semantic meaning of Cobol declarations. Possible types include numerics, alphanumerics and bynary-coded decimals, used for computational fields. The size of each variable is reported between square brackets; it is also split into integer and decimal parts for numerics (e.g. the numeric `B` field of record `R` has two integer and one decimal position: its type is therefore `num[2.1]`), and signed items are signed by a leading S character. Initialization values are also visible as assignments in this context.

The correctness of the retrieved information is easily verified. Note that, when each variable is referred in the actual code, the dot notation common to most modern programming languages eventually expresses it membership to a record, explicitly highlighting the hierarchy of declarations.

Ambiguities

Many ambiguities have been solved writing ad-hoc grammar productions. The most famous COBOL example is the so-called dangling-else problem, i.e. the uncertainty about which `IF` statement an `ELSE` clause belongs to. Like many similar situations, it can arise when nested constructs are used without specifying their optional delimiters (the `END_IF` keyword in this case). Consider the following line:

```
IF MY-VAR > 1 IF MY-BAR = 2 SUBTRACT 1 FROM MY-VAR ELSE DISPLAY MY-VAR.
```

It is unclear which statement the `ELSE` clause should be associated to. According to Cobol specification [4], an `ELSE` clause is paired with the nearest `IF` statement

that has no ELSE clause (the inner construct in the example). The Yacc grammar below obeys this rule:

```
statements:
   statement
 — statement statements

statement:
   ....
 — if_stmt
 — if_stmt END-IF

if_stmt:
   IF condition statements
 — IF condition THEN statements
 — IF condition statements ELSE statements
 — IF condition THEN statements ELSE statements
```

Figure 2.2: An IF-ELSE grammar for solving the dangling-else problem

As the reader may notice, the `if_stmt` productions have a common prefix. As explained in section 2.4.1, Yacc's policy is to shift over the input, trying to match the longest rule if possible: this yields the desired behavior. Numerous similar situations can be faced when developing a Cobol grammar; as we can see, Yacc allows to preserve a natural expression of grammar productions.

2.9.1 Project goals

A grammar specification for a task-focused subset of Cobol has been developed using Yacc, the currently most widespread parser generator. As far as the project goals are concerned, portions of input having a lower level of relevance are isolated even though their syntax is not specified, and a robust error handling mechnism assures tolerance to unsupported constructs. These two aspects are addressed in more detail in the reminder of this section.

Differently-relevant fragments

Thanks to context-aware lexing, the input is treated differently based on its relevance to type- and data-flow analysis. Consider the following:

```
IF MY-VALUE > 0 THEN
   DISPLAY POSITIVE VALUE: MY-VALUE
     WITH NO ADVANCING
   GO TO DO-SOMETHING
   OPEN OUTPUT A-FILE
END-IF.
```

Clearly, DISPLAY and OPEN are invariant statements. When Lex matches each of them, it generates an INVARIANT_KEYWORD token, and according to the grammar in figure 2.8.1, the entire construct is accepted using a catch-all production. Keywords appearing in it are distinguished from all other token classes by means of a general KEYWORD token. stream is shown below:

The role of the RESYNC token is crucial: thanks to it, Yacc is able to correctly tell statements apart (in particular, thoe added before the GO TO command and the END-IF delimiter mark the end of the two invariants in this example)

Error recovery

An error handling mechanism has been implemented, making the parsing process robust, i.e. tolerant with respect to constructs of unknown syntax, and at the same time keeping track of encountered symbols, which is useful for documentation purposes. Consider the following fragment:

```
IF MY-VAR > 3
    ADD WITH CONVERSION ID-1 TO ID-2
    ADD TO MY-VAR
    SOME UNRECOGNIZED STMT IS HERE
    MOVE FUNCTION ADD-DURATION(YYYYMMDD DAYS 90) TO MY-DATE
END-IF.
```

The many possible reasons of a missing parser specification for some input are herewith illustrated: an unsupported statement clause (e.g. WITH CONVERSION of the first nested statement), an error in the source file (e.g. the missing TO keyword in the second ADD), a completely foreign construct (as in SOME UNRECOGNIZED STMT IS HERE), or even unsupported language features (like intrinsic functions, used within the MOVE statement above). The fact that no separators might appear between each construct is an additional challenge: all delimiters, including the optional THEN and END-IF keywords of the IF statement, are missing.

The presented framework uses all information the source provides, including end of lines, and introduces missing delimiters where needed, producing a token stream reading which, thanks to the added RESYNC tokens, the parser is able to correctly separate each statement from the others. Exploiting COBOL syntactic structure and using lexical feedback, an entirely unknown construct is accepted even without entering error mode (SOME UNRECOGNIZED STMT IS HERE), and at the same time the presence of keywords is distinguished from other tokens.

An unsupported statement is displayed with leading question marks, followed by the list of identifiers appearing in it and by the corresponding input line, thanks to the token buffering mechanism described in section 2.8.1.

As a final remark, the large number of existing compiler implementations, language dialects and extensions is mentioned in section 2.5.1. Given the impossibility

of supporting all these different variants, one specific version was targeted. However, thanks to the robustness of the Procedure division parser, even in the case constructs from unsupported dialects are found, the framework is able to isolate them without compromising the parsing process. This is the most general possible solution: it allows support for common syntax, and at the same time is robust to inputs where foreign fragments appear.

2.9.2 Case study considerations

The developed parsing solution has been applied on legacy Cobol programs (attached to this document), for the purpose of testing its features in a real world setting.

For what concerns the project goals, we can state that they have been successfully achieved. At the moment of testing, the language coverage of the used grammar can be estimated to be approximately 35 %, including both supported and unrelevant syntax. No parse error was produced on the input, a remarkably significant fact taking into account the low percentage of Cobol support. This alone is a result that no other widespread parsing solution has been able to achieve to the writer's knowledge.

As for the quality of the output, due to the exact specification of interesting constructs, false positives cannot occur. Moreover, it was verified that no false negatives were produced as well, as the conditions mentioned in section 2.8.2 were met, yelding exactly the desired output.

Finally, some considerations on the generated syntax tree. Due to the differenes between Cobol and IL specifications (refer to section 3.1.3), the produced IL code is larger than the corresponding Cobol text in the number of statements (the number of lines is not a valid indicator in this case, due to the presence of multiple line statements, blank lines and comments in the input). An increase of about 140 % has been noticed, as branches, jumps and loop counters are introduced to mirror Cobol semantics.

2.10 Future work

Although the framework is able to deal with a remarkably large set of real world scenarios, one more feature is needed for dealing with all possible Cobol constructs. Post-processing of the parse tree for context-dependent ambiguities, mentioned in section 2.6.3, has not been implemented yet, and is planned for a further development stage.

A further thorough test phase is also planned, in order to confirm expected behaviour of the described parsing solution in a wider spectrum of scenarios. Instead of relying on a test suite of Cobol sources, which has to be maintained and expanded as further features are implemented, test cases could be generated in an automated fashion from the Cobol grammar definition in use. This way, all possible input

configurations could be easily tested; arbitrary syntax could also be randomly introduced among correctly generated constructs, in order to create fully comprehensive inputs for each grammar revision automatically.

Plans of future work also include an extension of the lexer with a functionality which is presently achieved manually. We refer to a special lexing mode, in which a general-purpose token (similarily to the `%token <token> ResyncKeyword` defined in section 2.8.1) is returned instead of that corresponding to the matched regular expression. A wrapper token type would be a very convenient implementation, as it would still allow access to the original token when needed. This would avoid the need of manually specifying a list of tokens legally accepted in the grammar catch-all production, thus simplifying maintenance of the framework as new token types are introduced.

3

Typing COBOL

3.1 Introduction

Analyzing COBOL code using type inference techniques has been proposed many times in the last decade and before. From the system first described in [77] to its later refinement in [56], giving informative types to COBOL variables seems to be a good way for automatically generating a basic tier of documentation of legacy software [81] and is also a reliable starting point for further Program Understanding approaches [46]. These systems are quite sophisticate and rely on a number of complex side models and tools aimed to extract properties and information from COBOL programs at a high level of abstraction, thus inevitably omitting several details at low level - e.g. how to deal exactly with the many picture formats supported by COBOL and with control constructs that alter the program flow.

Type analysis. Our proposal is a system for static analysis that is based on reconstructing types. The term *type reconstruction* is here used together with the verb *typing* for underlining an approach based on type rules aimed at understanding data inspecting procedure calls, constants, variable use and reuse etc[1]. No unification over type variables *à la ML* [25] occurs in our system and no abstract interpretation techniques take place: it is type reconstruction in the most general sense.

In this chapter we propose a revision of the system originally presented in [66] and selected for publication on [67]. It consists in a light-weight system for *statically analyzing* COBOL by means of custom types that pursue a number goals:

1. model the COBOL picture system without losing storage format information such as computational fields or the size of a numeric; this let us reconstruct the exact in-memory representation of datatypes and perform precise comparisons among the many formats COBOL supports;

2. deal with what in [80] is called *pollution* in such a way that no complex relational property system among types is needed, by tracking type alterations that variables are subject to in the following scenarios:

[1] In [62] the well-known ML type inference is referred to as a form of type reconstruction.

(a) when data is reused for different purposes in a program: many COBOL programmers have been used to this practice in order to save memory and the result is often poorly maintainable and error-prone code;

(b) when the language performs an implicit datatype cast, readapting value representation to fit a target variable having a different format than the source - and this can happen in COBOL either at compile-time or at run-time.

3. deal with branches in the program flow that are not statically decidable (i.e. conditional statements) by embedding into the type itself multiple types a variable may possibly assume during the execution. In other words, we don't try to guess how a condition is evaluated - we rather keep track of the types a variable might assume in multiple control-flow branches.

Type System.

Usually types are given both to expressions and computations that do not lie in memory and to data bound to variable names or memory cells. We introduce a form of type for both cases plus a higher-order special type for variables having multiple types.

1. *storage types* are the *single* types having an in-memory representation a variable may assume;

2. *temporary types* are the *single* types a computation can assume before it gets stored in memory or bound to a variable;

3. *choice types* are the *multiple* storage types a variable can assume resulting from conditional branches in the program flow;

4. *flow types* are simply a pair encapsulating a choice type and the original initialization storage type of a variable.

Jumps and Loops.

`GOTO` and `PERFORM` instructions are constantly found in legacy COBOL code as the main constructs for altering the control-flow of programs, hence our system cannot behave like an ordinary type-checker or type-inference algorithm: it looks more like a *type analyzer* capable of following jumps and branches in the code, detect cycles and avoid loops by checking for a convergence in topological environment (defined in section 3.1.5). This might resemble a weird form of abstract interpretation over types [31][2] but in fact is not: our system is a typing system performing type reconstruction; but it also recurs on `GOTO` statements until one of the environments does not change.

[2] Finding a convergence in the status of the typing function is almost trivial compared to interpreting computations on abstract domains. That is basically thanks to the static nature of types:

Implementation.

A prototype of the system described in this thesis has been developed in F# for the .NET 4.0 platform and includes a full-featured COBOL parser and translator to our *Intermediate Language* as well. A Lex & Yacc tweak has been designed for reproducing the behavior of Island Grammars [54] while keeping the benefits of an efficient LALR parser.

Typing-wise, it is currently able to parse large COBOL source programs (up to many hundreds of thousands of lines) and to type them generating as output the flow-types annotated at every variable occurrence. Additionally, it annotates useful information on type usage in form of error messages, warnings and hints. Again as opposed to a compiler, errors do not make the system fail: typing carries on and is tolerant to ill-typed situations by simply switching back to the initialization type of a variable whenever ambiguous scenarios are found.

3.1.1 Overview

Our system does not manipulate COBOL code directly: as other remarkable systems do [78], we translate COBOL into an *Idealized Language* (from now on referred to as IL) resembling modern imperative languages without altering COBOL semantics and principles. The syntax of IL is shown in section 3.1.3. Notably, what in COBOL speak is referred to as a program (i.e. a compilation unit), in IL becomes a procedure with its own set of static variable declarations. A COBOL application consisting of many programs translates into a single large IL program and the COBOL entry-point as its bottom unnamed block.

Before performing the type analysis, the system labels all variable occurrences in the program with an unique identifier - for example a fresh integer. The type analyzer eventually proceeds statement by statement and recursively descends into expressions, basically performing two operations:

1. updating the possible type changes a variable is subject to and keeping track of multiple types variables could concurrently assume;
2. annotating each variable occurrence with its current flow-type at that point in the program.

Assignments and call-by-reference argument passing are the two scenarios where variables could be subject to an implicit cast, hence the type of a variable could change. And conditional constructs are responsible for merging multiple types resulting from the typing of the two sub-blocks of an `if-then-else` statement into one *choice type*.

Look at the following sample code directly translated from COBOL into IL:

even including variable reuse, the overall number of reuses in any program is finite and can be statically determined in a finite number of passes.

```
{
  x := x + 1;
  if x > 0 then x := "foo";
  x := x + 23;
}
where x : num[2] := 11
```

Our system can reconstruct the types of the program and annotate each occurrence of variable `x` with its flow-type in that point of the code. Typing follows all branches in the control-flow: after the `if` block the system has to remember that `x` *might* have become a string. Also, where an ambiguous or ill-typed operation takes place, the system reports that and recovers to a default decision.

```
{
  (x : num[2]) := (x : num[2]) + 1;
    // [WARNING] possible truncation in assignment: num[3] :> num[2]

  if (x : num[2]) > 0 then
      (x : alpha[2]) := "foo";
    // [ERROR] truncation in assignment: alpha[3] :> num[2]

  (x : num[2]) := (x : num[2]|alpha[2]) + 23;
    // [HINT] x: ambiguity: assuming initialization type num[2]
    // [WARNING] possible truncation in assignment: num[3] :> num[2]
}
where x : num[2] := 11
```

In the statement at line 1, in which `x` gets incremented by 1, its type is annotated both at its usage as a right-value and as the target at the left hand of an assignment. In the right-hand its type is the initialization type `num[2]`, which obviously happens to be its current type at the beginning of the program; in the left-hand `x` should apparently be given a wider numeric type, because the result of the sum of a `num[2]` and a literal whose type is `num[1]` actually leads to a `num[3]`[3], but it gets truncated in order to fit the initialization type, exactly as COBOL run-time does. The final type happens to be equivalent to the initialization type and nothing seems changed, but internally the whole process has passed through the creation and the truncation of the temporary type `num[3]`.

Encountering the `if` statement makes the analyzer descend into the `then` block: a truncation is detected therein (being `alpha[3]` wider than the target type `num[2]`) and the truncated type `alpha[2]` is finally given to `x`, which fits the initialization type. Such information must then be merged to what had been previously inferred before the `if` statement: hence why, in the assignment under ther `if` block, the

[3]Simply because a number of 2 digits plus a number of 1 digit could possibly lead to a number of 3 digits, as $99 + 9 = 109$. See the type rules for expressions in table 3.7 for details on how arithmetic operations formally affect numeric type formats.

type of `x` in the right hand is not a simple type. A choice type has been introduced here by the merge: it shows the multiple types `x` might have at that program point. Which leads to an ambiguity when typing the sum operation and so the system needs to recover to the initial type declaration to prevent from failing. That might seem odd, but is in fact a viable solution: in COBOL every variable strictly adheres to its picture declaration, thus falling back to the initialization type is not unsafe - it is just inaccurate, but it serves just as a last resort in unsolvable situations.

3.1.2 Comparisons and Motivation

As already mentioned, the legacy software analysis system thoroughly presented in [56][4] rely on mechanisms for producing information over types that mainly serve Program Understanding techniques, Concept Analysis [46] and other high-level elaborations. In general, its scope is wider than ours and not entirely overlapping. While the basic goal may look similar, i.e. giving somehow interesting types to COBOL variables, there are several differences.

- We translate COBOL into a simpler intermediate language as [77] does, though without leaving out important language constructs whose behavior is relevant to typing real-world programs, such as jump primitives, procedure calls and conditional statements.
- Our type syntax is more complete and open to orthodox type manipulation, as it doesn't provide a flat representation of COBOL picture system[5].
- The type inference rules given in [80][6] are sometimes a tad trivial. In our type system the type reconstruction is more detailed, e.g. our type rules for arithmetic operators in table 3.7 recalculate the resulting type format length in order to detect a number of size errors at typing time.
- We don't infer a type equivalence when two or more types are expected to be the same. Our system rather falls back to a variable initialization type in case a type mismatch or ambiguity is detected. This trade off does not necessarily imply a loss of information and reflects COBOL run-time semantics better.

[4]That is a Ph.D. thesis collecting previous works on the same subject and anticipating some that yet had to come. In general, that system has been proposed several times in more articles with some additions - we might therefore refer to either [77], [80], [79], [46] or [56].

[5]Syntax of types in [77], weirdly, include variable names and picture format strings into type terms, leaving unclear how the type environment and type comparison functions handle them.

[6]The word *inference*, with a clear reference to ML and type theory in the functional language world, is a bit improper for the context: we'd rather prefer *reconstruction* or *analysis*, as there is no use of type variables and unification for resolving a set of constraints over type equations as in actual type inference systems [25].

- The system in [46] represents the inferred set of type relations via a Relational Algebra and resolves it by applying an algorithm written in Grok [38]: the resolution is actually a *simplification* process performing iterative unification. This approach doesn't seem to take into account control-flow jumps. Our system performs a code analysis at typing time by following branches and jumps and thus detects a wider range of possible type anomalies and variable reuses.
- According to [80], *pollution* occurs whenever a type-equivalence involves types that are not equivalent or subtypes: we do not handle this as a special case, but it automatically comes from *non-singleton choices* within flow-types, which are natively supported by our type-system and do not require any further processing.
- Our *subtype* relation deals with the in-memory representation of a wider range of type formats and qualifiers that are very common in COBOL programs, such as all `COMP` fields (translated into native integer, floating point and binary-coded-decimal types), signed/unsigned numeric formats and mixed alphabetic/alphanumeric strings.
- In [80] there is no mention on how COBOL references[7] are handled, nor on how COBOL run-time data conversions affect type rules manipulating different picture formats and computational fields (e.g. a `COMPUTE` instruction using mixed numeric variables and literals). A major feature of our system is to statically reproduce COBOL run-time and compile-time behaviors, keeping track of numeric formats and sizes and introducing *temporary types* for R-values[8] which are eventually *promoted* to storage types when a type coercion to a L-value occurs (see definitions 3.1.1 and 3.1.6).

3.1.3 Idealized Language Syntax

Grammar rules for IL syntax are given below together with lexical rules for identifiers, literals and operators. Terminal symbols are underlined, non-terminal symbols are in italics and EBNF meta-operators are in plain form.

3.1.4 Storage Types and Flow-Types

COBOL picture declarations in the Working Storage section of the Data Division define data instances along with their own storage format: they're not just type dec-

[7] According to COBOL syntax specification in [39], accessible memory cells are called *references*. We renamed them as *left values* in our intermediate language for the sake of symmetry with imperative languages such as C that define them as a sub-class of expressions which can appear at the left side on an assignment and refer to a real memory location [43].

[8] Symmetrically, emphright values are expressions that can stand on the right side of an assignments, hence evaluate to a temporary value [68] that does not lie in memory.

Table 3.1: IL Syntax..

P	$:=$	$\underline{\texttt{proc}}\ p\underline{(}(y\ \underline{:}\ \tau)*\underline{)}\ B\ \underline{\texttt{in}}\ P$	**procedure**
	$\mid$	B	**main**
B	$:=$	$st\ \underline{\texttt{where}}\ (x\ \underline{:}\ \tau\ [\underline{:=}\ lit])*$	**body with environment bindings**
st	$:=$	$lv\ \underline{:=}\ e$	**assignment**
	$\mid$	$\underline{\texttt{if}}\ e\ \underline{\texttt{then}}\ st\ [\underline{\texttt{else}}\ st]$	**if-then/if-then-else**
	$\mid$	$p\underline{(}(a)*\underline{)}$	**call**
	$\mid$	$\underline{\texttt{goto}}\ l$	**goto**
	$\mid$	$\underline{\texttt{perform}}\ l\ [l]$	**perform/perform-thru**
	$\mid$	$\underline{\texttt{return}}$	**return**
	$\mid$	$[l\underline{:}]\ \underline{\{}\ (st)+\ \underline{\}}$	**anonymous/named-block**
lv	$:=$	x	**variable**
	$\mid$	$lv\underline{[}e\underline{]}$	**array subscript**
	$\mid$	$lv\underline{.}z$	**record field select**
a	$:=$	$\underline{\texttt{val}}\ e$	**call-by-value**
	$\mid$	$\underline{\texttt{ref}}\ lv$	**call-by-reference**
e	$:=$	$e\ (op_a \mid op_l \mid op_r)\ e$	**binary operator application**
	$\mid$	$(- \mid not)\ e$	**unary operator application**
	$\mid$	lit	**literal**
	$\mid$	lv	**l-value**
lit	$:=$	$[\underline{-}]n[\underline{.}n]$	**numeric literal**
	$\mid$	$\underline{"}\ [\hat{}\,"]*\ \underline{"}$	**string literal**
	$\mid$	$true \mid false$	**boolean literal**
x, y, z, l, p	$:=$	$[\underline{\texttt{a}}-\underline{\texttt{z}}\ \underline{\texttt{A}}-\underline{\texttt{Z}}][\underline{\texttt{a}}-\underline{\texttt{z}}\ \underline{\texttt{A}}-\underline{\texttt{Z}}\ \underline{-}\ \underline{_}\ \underline{\texttt{0}}-\underline{\texttt{9}}]*$	**identifiers**
op_a	$:=$	$\underline{+} \mid \underline{-} \mid \underline{*} \mid \underline{/}$	**binary arithmetic operators**
op_l	$:=$	$\underline{\texttt{and}} \mid \underline{\texttt{or}}$	**binary logic operators**
op_r	$:=$	$\underline{<} \mid \underline{=} \mid \underline{!=} \mid \underline{<=} \mid \underline{>} \mid \underline{>=}$	**binary relational operators**
n	$:=$	$[\underline{\texttt{0}}-\underline{\texttt{9}}]+$	**natural number**

larations. Our system must of course reproduce this design, but mapping COBOL picture format strings into types. For example, consider the following picture declaration:

```
DATA DIVISION.
WORKING-STORAGE SECTION.
        01 A PIC 9(3) COMP-3 OCCURS 10.
        01 N PIC COMP 9(8).
        01 R1.
                02 R1-S PIC A(2).
                02 R1-B PIC X(3)9(2)A(3).
    01 R2 OCCURS 7.
                02 X PIC S99V9 COMP-2.
```

We translate it into more readable and compact type bindings that are quite self-explanatory:

$$
\begin{array}{rcl}
A & : & num_{bcd}[3]\ array[10] \\
N & : & num_{int32}[8] \\
R1 & : & \{S : alpha[2];\ B : alphanum[8]\} \\
R2 & : & \{X : num_{float64}[S2.1]\}\ array[7]
\end{array}
$$

Picture format strings are mapped into either numeric, pure alphabetic or alphanumeric types according to their structure; arrays and records are also first-class citizens of the type language in our system and can therefore be nested at will, yielding to types that resemble those of modern functional languages. Moreover, numeric types carry along detailed information on their in-memory representation at machine level, sign and length of both integral and fractional parts; while arrays and alphabetic/alphanumeric strings simply carry their length. The full syntax of the type-system follows:

$$
\begin{array}{lcll}
\tau & := & & \textbf{storage types} \\
 & & num_q[\rho] & \textbf{numeric} \\
 & | & alpha[n] & \textbf{alphabetic string} \\
 & | & alphanum[n] & \textbf{alphanumeric string} \\
 & | & \tau\ array[n] & \textbf{array} \\
 & | & \{x_1 : \tau_1 \, .. \, x_n : \tau_n\} & \textbf{record} \\
 \\
\sigma & := & & \textbf{temporary types} \\
 & & \tau & \\
 & | & bool & \textbf{boolean} \\
 & | & \overline{num}[\rho] & \textbf{abstract numeric} \\
 \\
q & := & & \textbf{numeric storage qualifier} \\
 & & ascii & \textbf{display } \textbf{\textit{or}} \textbf{ ASCII} \\
 & | & bcd & \textbf{binary-packed decimal} \\
 & | & int_{16|32|64} & \textbf{native integer} \\
 & | & float_{32|64} & \textbf{native float} \\
 \\
\rho & := & [S]n.d & \textbf{numeric format} \\
 \\
\varphi & := & \{\tau_1 \, .. \, \tau_k\} & \textbf{flow-item } \textbf{\textit{or}} \textbf{ choice} \\
 \\
\Phi & := & \langle \varphi; \tau \rangle & \textbf{flow-type}
\end{array}
$$

$$\texttt{where } k \geq 1\texttt{, } n \in \mathbb{N}^*\texttt{, } d \in \mathbb{N}$$

There are two distinct classes of types:

- τ is the type of storage variables and L-values in general, i.e. the type of data that stands in memory and has some representation[9];
- σ, where $\sigma \supset \tau$, is the type given to expression terms only and is never produced by picture translation, serving just as a temporary light-weight type whose in-memory representation is yet to be known in that context.

As typing rules will show,such temporary types are eventually promoted to ordinary τ types as soon as the storage type of an actual variable becomes known, for example when an expression that's given a temporary is then assigned to a L-value or passed as a call-by-ref argument in an procedure call. Finally, a *flow-type* is a simply a pair of possibly multiple storage types (those a variable may concurrently have following statically undecidable conditional branches in the program flow, as stated in section

[9] ASCII is the default qualifier for numeric types: whenever unspecified this one holds, as in `num[3]` for example.

3.1) and an additional single storage type, which is the type initially declared for the variable in the global environment. We'll be often referring to the first component of a flow-type as *flow-item* or *choice*.

3.1.5 Environments

Type rules operate over a number of environments mapping different entities.

Type Environment Γ maps variable identifiers to flow-types: this environment is initially populated with global type declarations and its bindings are then updated when the current flow-type changes during typing. It contains bindings of form $x : \Phi$.

Topological Environment Θ collects all annotations produced by the type analyzer by mapping labeled variable occurrences x^κ to its flow-type at that program point. It represents also the status of the typing function in the detection of loop termination. It contains bindings of form $x^\kappa : \Phi$.

Procedure Environment Π maps procedure names to signatures (see definition 3.1.8). It contains bindings of form $p \mapsto \langle y_1 : \tau_1^p \;..\; y_n : \tau_n^p; \Gamma_p\rangle$.

Block Environment Σ maps label identifiers to blocks of statements. It contains bindings of form $l \mapsto \underline{\{} st_1 \;..\; st_n \underline{\}}$.

Type environments also support a binary function `merge`, used by rules If and If-Else, which recompacts the bindings collected in separate environments by the typing of program branches, as informally introduced by section 3.1. Such `merge` function is alone responsible for the growth of the flow-item component within a flow-type.

3.1.6 Coercion of L-Values

Take the following example:

```
{
a[0].l := "boo";
}
where a : { l : num[2]; m : alpha[10] } array[5]
```

And its annotated form resulting from the type analysis:

```
{
(a : { { l : alpha[2]; m : alpha[10] } array[5])[0].l
    := "boo";
}
where a : { l : num[2]; m : alpha[10] } array[5]
```

The literal `"boo"` having type `alpha[3]` is assigned to field `l` of a record within a cell of an array. The flow-type of variable `a` needs to be updated here somehow with the type of the right-hand of the assignment - and of course it's not to `a` that such type must naively be given, but to the record field `l` nested within. Nonetheless the environment binds variable identifiers to flow-types, thus there is no way to update the type of a record label (as `l` in our case) or of an array cell alone. Therefore the whole type of a variable must be updated keeping the original structure layout and replacing the appropriate bit nested within it. Hence, the whole type of `a` in the example becomes { `l : alpha[2]; m : alpha[10]` } `array[5]`.

This shows also that the expected type `alpha[3]` of the literal `"boo"` has been adapted to *fit* into the initialization type `num[2]`: coercion in assignments needs therefore both to replace a piece of a type and to resize it accordingly, keeping the original storage class (`num` in our example) and recalculating the format in such a way that the overall size of the new resulting type fits the initialization one.

For this reasons, judgements for L-value terms are slightly different: $\Pi; \Sigma; \Gamma; \Theta_0 \vdash_{lv} lv : \tau \backslash \theta^{x^\kappa} \; \triangleright \; \Theta_1$ means that the L-value lv has a storage type τ coercible by the substitution θ^{x^κ}, where x is the root variable of lv (formally $x = \Re(lv)$ as of definition 3.1.7) and x^κ is its labeled occurrence. θ is a function from storage types to storage types that can be passed by typing rules that need to update the type of the root variable of an L-value to the coerce function $\mathscr{C}$ (see definition 3.1.6), which performs the proper fit operation among other things.

3.1.7 Loops and Convergence

As informally stated in section 3.1, the type analyzer follows `goto` and `perform` statements unless already visited and a convergence in the status of the typing function is detected. In subsection 3.1.5 we said that this status actually consists of the topological environment Θ. The typing function at step i of the analysis can be defined as a function taking the statement fetched at that step and the topological environment:

$$_i(st_{B,p}, \Theta_i) = \Theta_{i+1}$$

where $st_{B,p}$ is the statement located within block B at position p.

Each time the typing function encounters a jump statement, it performs a number of operations. Say a jump statement $st_{A,q} \equiv$ `goto` l is encountered by at step i while typing block $A = \{st_{A,1} \;..\; st_{A,n}\}$ (with $q \in [1, n]$):

1. it saves the topological environment Θ_i built up so far, binding it to the current program location;

2. it looks up the destination block of statements from the block environment, hence $B = \{st_{B,1} \;..\; st_{B,m}\} = \Sigma(l)$;

3. it continues the analysis from there, i.e. from statement $st_{B,1}$.

Let's consider that later at step j (obviously $j > i$) reaches the jump statement $st_{A,q}$ again: then the new current topological environment Θ_j is compared against Θ_i, which had formerly been saved at that program location. If $\Theta_j \sqsubseteq \Theta_i$ (see definition 3.1.10) then it means that no further type information has been collected during the second pass and we can therefore assume that the analysis can safely skip the jump statement $st_{A,q}$ and continue from $st_{A,q+1}$. Else, the new topological environment Θ_j is saved (replacing the old Θ_i previously stored) and the analysis continues from the jump statement destination $st_{B,1}$ again.

We observed that the system detects a convergence averagely in 1 and anyway in up to 3 reiterations of the same piece of code. The reason is twofold:

- the topological environment cannot by definition be subject to binding removal, hence $\forall x^\kappa \in \Theta_i.\ x^\kappa \in \Theta_{i+1}$ at any given step i;
- flow-types bound to variable occurrences in the topological environment can only grow - they can never diminish in width. Given we're dealing with types and not values, the stability is certain: storage types of variables do not change from pass to pass for obvious reasons and the only thing that could change and modify the status Θ of the typing function is the flow-item φ part of flow-types bound to variable occurrences. φ is defined as a set of storage types τ in table 3.1.4 and it is subject to a single operation: the `merge` function as of definition 3.1.9, which basically consists in a set-union between flow-items. Duplicate types can therefore never occur and no element could be removed.

3.1.8 Ambiguity

Having non-singleton flow-items within flow-types is indeed a central feature of this system, signaling that the programmer reused a variable in different ways along the program. Nonetheless, that makes judgments for L-values problematic: how are we supposed to type an L-value appearing in an expression, for instance, if its current flow-type says that it could have many storage types at the same time? In fact, we can't - that's exactly what flow-types stand for: detecting anomalous scenarios that may lead to unwanted results at run-time.

In our code example in section 3.1, imagine the system had output another hint message for the ambiguous statement claiming that among the possible choices `num[3]` would have been suitable. And the typing then proceeded selecting `num[3]` as candidate, leading to a different type for `x` - not the one shown in the original example.

```
{
(x : num[3]) := (x : num[2]) + 1;
// [WARNING] possible truncation detected
//           in assignment:
//              num[3] :> num[2]
```

```
if (x : num[3]) > 0 then
{
    (x : alpha[3]) := "foo";
    // [ERROR] truncation detected in
    //         assignment:
    //          alpha[3] :> num[2]
}

(x : num[6]) := (x : num[3]|alpha[3]) + 23;
// [HINT] type of 'x' is ambiguous in
//        expression at right-hand of
//        assignment: choice num[3]
//        would fit
// [WARNING] possible truncation detected
//           in assignment:
//              num[6] :> num[2]
}
where x : num[2] := 11
```

What if more than one type was suitable, though? What type would the whole expression - and therefore `x` - have had then? Arguably multiple types resulting from different typing paths - but if so, which path led to which result type? The flow-type would literally explode for tracking several implications among possible typing paths and in the end it would hardly be useful.

Our proposal in such situations is to do the simplest thing: falling back to the initial type of the variable; and of course notifying the choice with a hint message. However, this leads to a duplication of the type rule for variables, as table 3.5 shows.

In this section we give the full specification of the IL language and the type-system described in section 3.1.4.

3.1.9 Definitions

A number of formal definitions is needed before presenting type rules.

Definition 3.1.1 (Promote)**.** The promotion $[\![\sigma]\!]^\tau$ of a temporary type σ to a storage type τ produces a storage type that transform σ into a storable type inheriting the characteristics of τ. The promotion function is defined as follows (top-down closest-match rule on the left hand holds):

$$\begin{array}{rclrcl} [\![\overline{num}[\rho_2]]\!]^{num_q[\rho_1]} & = & num_q[\rho_2] & [\![\overline{num}[\rho]]\!]^{\tau} & = & num_{ascii}[\rho] \\ [\![bool]\!]^{\tau} & = & [\![\overline{num}[1.0]]\!]^{\tau} & [\![\tau_2]\!]^{\tau_1} & = & \tau_2 \end{array}$$

Notice, though, that the case where a *seq* type is promoted to an *array* cannot actually occur in any real-world scenario since COBOL, as well as IL, does not

support array literals in its expression syntax but only in default-value definitions within data declarations (i.e. pictures in the data division translated into IL as environment bindings specifying a sequence of literal constants).

Definition 3.1.2 (Representation)**.** We define a function $\mathtt{rep} : \tau \to \mathbb{N}$ for calculating the in-memory byte size of a storage type:

$$\begin{aligned} \mathtt{rep}(num_{ascii}[n.d]) &= n + d \\ \mathtt{rep}(num_{bcd}[n.d]) &= \lceil \tfrac{n+d+1}{2} \rceil \\ \mathtt{rep}(num_{int_b}[\rho]) &= b/8 \\ \mathtt{rep}(num_{float_b}[\rho]) &= b/8 \end{aligned} \qquad \begin{aligned} \mathtt{rep}(alpha[n]) &= n \\ \mathtt{rep}(alphanum[n]) &= n \\ \mathtt{rep}(\tau\ array[n]) &= \mathtt{rep}(\tau) * n \\ \mathtt{rep}(\{x_1 : \tau_1..x_n : \tau_n\}) &= \sum_{i=1}^{n} \mathtt{rep}(\tau_i) \end{aligned}$$

Definition 3.1.3 (Subtype)**.** We define a total-order between storage types such that the relation $\tau_1 \preceq \tau_2$ holds when $\mathtt{rep}(\tau_1) \leq \mathtt{rep}(\tau_2)$.

Definition 3.1.4 (Var-Bound Substitution)**.** A substitution θ^{x^κ} is a function from storage types to storage types that carries along a labeled identifier x^κ which stands for the variable occurrence whose type the substitution has been built from and is supposed to replace[10].

Definition 3.1.5 (Fit)**.** The fit $\lfloor \tau_1 \rfloor_{\tau_2}$ of a storage type τ_1 to a storage type τ_2 produces a storage type whose storage class is equivalent to that of τ_1 and whose size fits into that of τ_2. The fit function is defined as follows:

$$\begin{aligned} \lfloor num_q[\rho] \rfloor_\tau &= num_q[\rho'] & \mid\ & \mathtt{rep}(num_q[\rho']) = \mathtt{rep}(\tau) \\ \lfloor alpha[n] \rfloor_\tau &= alpha[n'] & \mid\ & \mathtt{rep}(alpha[n']) = \mathtt{rep}(\tau) \\ \lfloor alphanum[n] \rfloor_\tau &= alphanum[n'] & \mid\ & \mathtt{rep}(alphanum[n']) = \mathtt{rep}(\tau) \\ \lfloor \tau_a\ array[n] \rfloor_\tau &= \tau'_a\ array[n'] & \mid\ & \mathtt{rep}(\tau'_a\ array[n']) = \mathtt{rep}(\tau) \\ \lfloor \{l_1 : \tau_1..l_n : \tau_n\} \rfloor_\tau &= \{l_1 : \tau'_1..l_n : \tau'_n\} & \mid\ & \mathtt{rep}(\{l_1 : \tau'_1..l_n : \tau'_n\}) = \mathtt{rep}(\tau) \end{aligned}$$

Definition 3.1.6 (Coerce)**.** The coerce function $\mathscr{C}$ updates the given type and topological environments by applying a given substitution function θ^{x^κ} to the types a given flow-item φ consists of; it produces a new pair of form $\langle \Gamma; \Theta \rangle$ consisting of the type and topological environments endowed with updated bindings for the variable x and the occurrence label κ annotated on the substitution function θ^{x^κ} itself:

$$\mathscr{C}(\varphi, \theta^{x^\kappa}, \Gamma, \Theta) \;=\; \langle \Gamma, x : \Phi'; \Theta, \kappa : \Phi' \rangle \quad with \quad \begin{array}{l} \langle \varphi; \tau_x \rangle = \Gamma(x) \\ \Phi' = \langle \{\forall \tau_i \in \varphi. \lfloor \theta^{x^\kappa}(\tau_i) \rfloor_{\tau_x}\}; \tau_x \rangle \end{array}$$

[10]Substitution functions are recursively defined by type rules for L-Values as shown in table 3.5. They're meant for generically replacing a term nested within a storage type of arbitrary complexity by reproducing its original structure of recursive type terms and changing the innermost part only.

Definition 3.1.7 (Root Variable)**.** Given an L-value lv, its root variable is the identifier x evaluated by the recursive function defined as:

$$\Re(x) \;=\; x \qquad \Re(lv[e]) \;=\; \Re(lv) \qquad \Re(lv.l) \;=\; \Re(lv)$$

Definition 3.1.8 (Signature)**.** A signature is a pair $\langle Y_p; \Gamma_p \rangle$ where p is a procedure name, Y_p are its formal parameters $y_1 : \tau_1^p..y_n : \tau_n^p$ and Γ_p is the output type environment returned by typing the body of p.

Definition 3.1.9 (Type Environment Merge)**.** The binary function $\oplus$ merges two given type environments into one as $\Gamma_1 \oplus \Gamma_2 = \Gamma^* \cup (\Gamma_1 \backslash \Gamma_2) \cup (\Gamma_2 \backslash \Gamma_1)$ where $\Gamma^* = \{x : \langle \varphi_1 \cup \varphi_2; \tau_1 \rangle | \Gamma_1(x) = \langle \varphi_1; \tau_1 \rangle \wedge \Gamma_2(x) = \langle \varphi_2; \tau_2 \rangle \wedge \tau_1 = \tau_2\}$.

Definition 3.1.10 (Partial Ordering of Flow-Types)**.** We define a partial order between flow-types such that $\Phi_1 \sqsubseteq \Phi_2$ holds when, let $\Phi_1 = \langle \varphi_1; \tau_1 \rangle$ and $\Phi_2 = \langle \varphi_2; \tau_2 \rangle$, then $\varphi_1 \subseteq \varphi_2 \wedge \tau_1 = \tau_2$.

Definition 3.1.11 (Partial Ordering of Topological Environments)**.** We define a partial order between topological environments such that $\Theta_1 \sqsubseteq \Theta_2$ holds when $\forall x : \Phi_1 \in \Theta_1.x \in dom(\Theta_2) \wedge \Phi_1 \sqsubseteq \Phi_2$, where $\Phi_2 = \Theta_2(x)$.

3.1.10 Properties of the System

One important property of our system is termination: since GOTO statements produce a recursion in our typing rules, one question about terminability arises.

Proposition 3.1.1 (Termination)**.** *For any COBOL program represented by an IL program P, the type judice $\emptyset; \Theta_0 \vdash_P P \;\triangleright\; \Theta_1$ implies a finite derivation.*

A real proof would require a lot of additional formal tools, including an additional map or environment in which already visited GOTO occurrences in the program are stored. That would make our type rules even more complicated for gaining a property that is quite straightforward to imagine. Intuitively, sooner or later the topological environment will be populated with all variable occurrences of every reachable code block, thus a convergence will be found in a relatively low number of passes through the same given GOTO occurrence. Convergence is defined of course as the topological environment not changing anymore (as of definition), and since we're speaking of types here and not values, changes may happen only in conjunction with non-unary flow types, which in turn are created only by the IF rule - i.e. in program branches which are of course finite.

Another interesting property would be some kind of soundness. We cannot deliver detailed proofs in this thesis due to a number of reasons, among which the fact that that would require a full formal semantics for COBOL, which has never been done in a satisfying way in literature, being COBOL such a complex language. We can formulate a form of weak subject reduction, though. We claim that at, the

end of the type analysis, for every variable occurrence mapped to its flow type by the topological environment, the choice type part within such flow types will contain all storage types that that variable can possibly assume in that program point at runtime; and all such storage types do fit into the variable initial type. We need to introduce the notion of semantics, reduction and values, though, to achieve some degree of formal precision. Arguably, most of the semantics of IL should be pretty trivial, being a generic imperative language resembling COBOL: it behaves like any ordinary language with assignments and static scopes. Complex reduction rules would probably be arithmetic operators and data conversion in assignments, though these are beyond our current scope. Since we're interested in variable occurrences, we should only care about the rule for variable lookup. In an operational form, that shall be:

Definition 3.1.12 (Reduction Rule for Variable Lookup)**.** The following rule holds for variables appearing as right-values in a program: $\dfrac{x \in \Delta}{\Delta \vdash x^{\kappa} \Downarrow v}$ where v is a value represented by an IL literal defined by the non-terminal *lit* in syntax table 3.1, and Δ is an environment mapping identifiers to values.

Notably, x - and not x^{κ} - belongs to Δ , because only variable plain names are mapped by the environment, ignoring the unique label annotated on top of it. Also it's easy to imagine that Δ gets populated when entering a procedure scope and its bindings are rebound in case of assignments.

Now let us formulate a proposition for guaranteeing some form of subject reduction for variable occurrences:

Proposition 3.1.2 (Weak Subject Reduction)**.** *Given a COBOL program represented by an IL program P, let Θ_1 be the topological environment output by the type judice $\emptyset; \Theta_0 \vdash_P P \ \triangleright \ \Theta_1$. For any given labeled variable identifier x^{κ} occurring as a right-value in P, let x^{κ} evaluate to a value v according to the reduction rule in definition 3.1.12, then $\exists \tau_i \in \varphi \mid v : \tau_i$, where $x^{\kappa} : \langle \varphi; \tau_0 \rangle \in \Theta_1$ and $\varphi = \{\tau_1..\tau_n\}$. Additionally, $\lfloor \tau_i \rfloor_{\tau_0}$.*

The proposition above claims two important safety properties:

- for any given variable occurrence in a program, a storage type exists within the reconstructed flow type for each possible value that that variable could evaluate to at runtime; and
- such storage type does fit (see definition 3.1.5) into the declared initial storage type of that variable, therefore any value hosted by the variable would not imply data corruption or memory issues.

A realistic proof would require the complete operational semantics definition for IL, as said, but an arguably straightforward proof sketch can be anticipated. The

type of the value v of a given variable occurrence x^κ is present in the flow type $\langle \varphi; \tau_0 \rangle$ bound to x^κ in the the topological environment Θ_1 because that v had to be assigned to x earlier in the program in a way or another: if no branches in the code block occur, then trivially rule ASSIGN in table 3.3 shows how the type bound to x in the type environment Γ is replaced with the type of the right-value, i.e. inductively the type of v - and Θ_1 is populated with every single type of x for every occurrence x^κ throughout the program as rule VAR-CURR shows. Else, in case code branches occur, rule IF is the only responsible for the creation of non-unary choice types within flow types (via the environment merge operation as of definition 3.1.9), therefore every possible assignment within `then` or `else` blocks brings either to the former trivial case in case no nested conditionals occur or to the latter case inductively.

3.1.11 Type Rules

Syntax-directed type rules are divided by category. Rules for Programs are shown in table 3.2, for Statements in table 3.3, for Expressions in table 3.7, for Arguments in table 3.4 and for Literals in table 3.6.

Most judgements give a type to a term of the language in a context consisting of a tuple of environments and output the updated Γ and Θ, except judgements for Statements and Programs that give no type and simply update the environments. As a general rule, the topological environment Θ is always forwarded to and returned by all judgements (except literals), because flow-types must be annotated recursively on each variable occurring in any subterm of the program. While the type environment Γ is output only by rules that actually update it: consider it as returned back untouched when there's no mention of it among outputs.

Table 3.2: Type Rules for Programs and Body.

MAIN

$$\frac{\Pi; \emptyset; \Theta_0 \vdash_B B \ \triangleright \ \Gamma; \Theta_1}{\Pi; \Theta_0 \vdash_P B \ \triangleright \ \Theta_1}$$

PROC

$$\frac{\begin{array}{c}\Gamma_p = \emptyset, y_1 : \langle \{\tau_i^p\}; \tau_i^p \rangle .. y_n : \langle \{\tau_n^p\}; \tau_n^p \rangle \\ \Pi; \Gamma_p; \Theta_0 \vdash_B B \ \triangleright \ \Gamma'_p; \Theta_1 \qquad \Pi, p \mapsto \langle y_1 : \tau_1^p .. y_n : \tau_n^p; \Gamma'_p \rangle; \Theta_1 \vdash_P P \ \triangleright \ \Theta_2\end{array}}{\Pi; \Theta_0 \vdash_P \underline{\texttt{proc}}\ p(y_1 : \tau_1^p .. y_n : \tau_n^p)\ B\ \underline{\texttt{in}}\ P \ \triangleright \ \Theta_2}$$

BODY

$$\frac{\begin{array}{c}\forall i \in [1,n].\Pi; \Sigma; \Gamma_0; \Theta_0 \vdash_{lit} lit_i : \sigma_i \wedge [\![\sigma_i]\!]^{\tau_i} \preceq \tau_i \\ \Pi; \emptyset; \Gamma_0, x_1 : \langle \{\tau_1\}; \tau_1 \rangle .. x_n : \langle \{\tau_n\}; \tau_n \rangle; \Theta_0 \vdash_{st} st \ \triangleright \ \Gamma_1; \Theta_1\end{array}}{\Pi; \Gamma_0; \Theta_0 \vdash_B st\ \underline{\texttt{where}}\ x_1 : \tau_1 := lit_1 .. x_n : \tau_n := lit_n \ \triangleright \ \Gamma_1; \Theta_1}$$

Judgements are of a number of forms, each syntactic category having its own, though most of them are quite self-explanatory. For example, $\Pi; \Sigma; \Gamma; \Theta_0 \vdash_e e : \sigma \triangleright \Theta_1$ denotes that, in the given environments, expression e is given a temporary type σ and the topological environment Θ_1 is output.

Judgements for Arguments probably need some extra words. Call-by-ref calls need to update the type environment of the the caller because the flow-type of argument might be modified by the invoked procedure. The procedure environment Π stores the type environment Γ_p for each procedure p of the program, thus the flow-type of a variable passed by reference to p can be updated according to the flow-type of the corresponding formal parameter bound in Γ_p. Such update is carried on by the coerce function $\mathscr{C}$, as shown by rule BYREF in table 3.4. The mechanism resembles that in rule ASSIGN in table 3.3: call-by-reference argument application indeed behaves like an assignment (call-by-value doesn't).

Rules for Arguments have form $\Pi; \Sigma; \Gamma_0; \Theta_0 \vdash_a a : \tau_i \triangleright \Gamma_1; \Theta_1$, meaning that, in the given environments, the actual argument a has type τ_i^p, which is the type of the i-th formal parameter of procedure p.

As a final notice, for the sake of simplicity we assume that all labels in the program are named in order of occurrence: if l_n and l_m are two labels and $m > n$, then l_m appear *below* l_n in the program. That makes type rules for jump statements simpler.

3.2 Experimental Results

Our implementation of the system is the `COBOL Analyzer` tool and is available for download as anticipated in the preface. It can detect a number of type misuses and mismatches besides producing flow-type annotations for each variable occurrence. At the time of writing several tests have been run over real-world legacy business code. Being COBOL an old language, though, it is not possible to check full programs because they consist in millions of lines spread over thousands of modules: we rather tested code portions that we believed meaningful. All of them were written in COBOL85 for z/OS throughout the 1990s and owned by a big company in northern Italy within the mechanical vehicle industry.

They kindly sent us several unrelated code excerpts: some smaller than 10 KB and some larger than 50 MB. Admitedly, memory has been an issue during the tests, especially for parsing such long programs and keeping the IL AST representation in memory. Typing itself has never been a problem, instead, which probably means that the F# garbage collector (i.e. .NET's) work well even with heavily monadic code as well. Refer to section 4.6 for an in-depth view of some interesting features of the implementation, as both the COBOL and the Android analyzers share several design choices and patterns.

As far as results are concerned, the following considerations on type usage in COBOL programs have emerged:

Table 3.3: Type Rules for Statements.

ASSIGN

$$\frac{\Pi;\Sigma;\Gamma_0;\Theta_0 \vdash_e e : \sigma_e \ \triangleright\ \Theta_1 \qquad \Pi;\Sigma;\Gamma_0;\Theta_1 \vdash_{lv} lv : \tau_{lv}\backslash\theta^{x^\kappa} \ \triangleright\ \Theta_2 \quad x^\kappa = \Re(lv) \qquad \langle\Gamma_1;\Theta_2\rangle = \mathscr{C}([\![\sigma_e]\!]^{\tau_{lv}}, \theta^{x^\kappa}, \Gamma_0, \Theta_2)}{\Pi;\Sigma;\Gamma_0;\Theta_0 \vdash_{st} lv := e \ \triangleright\ \Gamma_1;\Theta_2}$$

IF

$$\frac{\Pi;\Sigma;\Gamma_0;\Theta_0 \vdash_e e : bool \ \triangleright\ \Theta_1 \qquad \Pi;\Sigma;\Gamma_0;\Theta_1 \vdash_{st} st \ \triangleright\ \Gamma_1;\Theta_2 \qquad \Gamma_2 = \Gamma_0 \oplus \Gamma_1}{\Pi;\Sigma;\Gamma_0;\Theta_0 \vdash_{st} \underline{\texttt{if}}\ e\ \underline{\texttt{then}}\ st_1 \ \triangleright\ \Gamma_2;\Theta_2}$$

IF-ELSE

$$\frac{\Pi;\Sigma;\Gamma_0;\Theta_0 \vdash_e e : bool \ \triangleright\ \Theta_1 \quad \Pi;\Sigma;\Gamma_0;\Theta_1 \vdash_{st} st_1 \ \triangleright\ \Gamma_1;\Theta_2 \qquad \Pi;\Sigma;\Gamma_0;\Theta_2 \vdash_{st} st_2 \ \triangleright\ \Gamma_2;\Theta_3 \qquad \Gamma_3 = \Gamma_1 \oplus \Gamma_2}{\Pi;\Sigma;\Gamma_0;\Theta_0 \vdash_{st} \underline{\texttt{if}}\ e\ \underline{\texttt{then}}\ st_1\ \underline{\texttt{else}}\ st_2 \ \triangleright\ \Gamma_3;\Theta_3}$$

PERFORM

$$\frac{\underline{\{}st_1..st_n\underline{\}} = \Sigma(l) \qquad \Pi;\Sigma;\Gamma_0;\Theta_0 \vdash_{st} \underline{\{}st_1..st_n\underline{\}} \ \triangleright\ \Gamma_1;\Theta_1}{\Pi;\Sigma;\Gamma_0;\Theta_0 \vdash_{st} \underline{\texttt{perform}}\ l \ \triangleright\ \Gamma_1;\Theta_1}$$

PERFORM-THRU

$$\frac{\forall i \in [a,b) \quad \underline{\{}st_{i,1}..st_{i,n_i}\underline{\}} = \Sigma(l_i) \qquad \Pi;\Sigma;\Gamma_{i-a};\Theta_{i-a} \vdash_{st} \underline{\{}st_{i,1}..st_{i,n_i}\underline{\}} \ \triangleright\ \Gamma_{i-a+1};\Theta_{i-a+1}}{\Pi;\Sigma;\Gamma_0;\Theta_0 \vdash_{st} \underline{\texttt{perform}}\ l_a\ l_b \ \triangleright\ \Gamma_{b-a-1};\Theta_{b-a-1}}$$

GOTO

$$\frac{l_n \in dom(\Sigma) \mid \nexists l_m.\ m > n \quad \forall i \in [k,n].\ \Pi;\Sigma;\Gamma_{i-k};\Theta_{i-k} \vdash_{st} \Sigma(l_i) \ \triangleright\ \Gamma_{i-k+1};\Theta_{i-k+1}}{\Pi;\Sigma;\Gamma_0;\Theta_0 \vdash_{st} \underline{\texttt{goto}}\ l_k \ \triangleright\ \Gamma_{n-k};\Theta_{n-k}}$$

CALL

$$\frac{\langle y_1 : \tau_1^p..y_n : \tau_n^p;\Gamma_p\rangle = \Pi(p) \qquad \forall i \in [1,n].\Pi;\Sigma;\Gamma_{i-1};\Theta_{i-1} \vdash_a a_i : \tau_i^p \ \triangleright\ \Gamma_i;\Theta_i}{\Pi;\Sigma;\Gamma_0;\Theta_0 \vdash_{st} p\underline{(}a_1..a_n\underline{)} \ \triangleright\ \Gamma_n;\Theta_n}$$

BLOCK

$$\frac{\forall j | st_{0,j} \equiv l_j\underline{:}\underline{\{}st_{j,1}..st_{j,n_j}\underline{\}} \qquad \Sigma' = \Sigma, l_j \mapsto \underline{\{}st_{j,1}..st_{j,n_j}\underline{\}}.. \quad \forall i \in [1,m] | m \leq n \wedge st_i \not\equiv \underline{\texttt{goto}}\ l \qquad \Pi;\Sigma';\Gamma_{i-1};\Theta_{i-1} \vdash_{st} st_i \ \triangleright\ \Gamma_i;\Theta_i}{\Pi;\Sigma;\Gamma_0;\Theta_0 \vdash_{st} l_0 : \underline{\{}st_{0,1}..st_{0,n_0}\underline{\}} \ \triangleright\ \Gamma_n;\Theta_n}$$

Table 3.4: Type Rules for Arguments.

ByVal

$$\frac{\Pi;\Sigma;\Gamma_0;\Theta_0 \vdash_e e : \sigma \ \triangleright\ \Theta_1 \qquad [\![\sigma]\!]^{\tau_i^p} \preceq \tau_i^p}{\Pi;\Sigma;\Gamma_0;\Theta_0 \vdash_a \underline{\texttt{val}}\ e : \tau_i^p \ \triangleright\ \Gamma_0;\Theta_1}$$

ByRef

$$\frac{\begin{array}{c}\Pi;\Sigma;\Gamma_0;\Theta_0 \vdash_{lv} lv : \tau'\backslash\theta^{x^\kappa} \ \triangleright\ \Theta_1 \qquad x = \Re(lv) \qquad \tau' \preceq \tau_i^p \\ \langle y_1 : \tau_1^p..y_n : \tau_n^p; \Gamma_p\rangle = \Pi(p) \qquad \langle \varphi_i^p; \tau_i^p\rangle = \Gamma_p(y_i) \qquad \langle \Gamma_1;\Theta_2\rangle = \mathscr{C}(\varphi_i^p, \theta^{x^\kappa}, \Gamma_0, \Theta_1)\end{array}}{\Pi;\Sigma;\Gamma_0;\Theta_0 \vdash_a \underline{\texttt{ref}}\ lv : \tau_i^p \ \triangleright\ \Gamma_1;\Theta_2}$$

Table 3.5: Type Rules for L-Values.

Var-Init

$$\frac{\Gamma(x) = \Phi = \langle\{\tau_1\tau_2..\tau_n\};\tau_0\rangle \qquad \Theta_1 = \Theta_0, x^\kappa : \Phi \qquad \theta^{x^\kappa}(\overline{\tau}) = \overline{\tau}}{\Pi;\Sigma;\Gamma;\Theta_0 \vdash_{lv} x^\kappa : \tau_0\backslash\theta^{x^\kappa} \ \triangleright\ \Theta_1}$$

Var-Curr

$$\frac{\Gamma(x) = \Phi = \langle\{\tau_1\};\tau_0\rangle \qquad \Theta_1 = \Theta_0, x^\kappa : \Phi \qquad \theta^{x^\kappa}(\overline{\tau}) = \overline{\tau}}{\Pi;\Sigma;\Gamma;\Theta_0 \vdash_{lv} x^\kappa : \tau_1\backslash\theta^{x^\kappa} \ \triangleright\ \Theta_1}$$

Subscript

$$\frac{\begin{array}{c}\Pi;\Sigma;\Gamma;\Theta_0 \vdash_e e : \overline{num}[\rho] \ \triangleright\ \Theta_1 \qquad \Pi;\Sigma;\Gamma;\Theta_1 \vdash_{lv} lv : \tau\ array[n]\backslash\theta_{lv}^{x^\kappa} \ \triangleright\ \Theta_2 \\ x = \Re(lv) \qquad \theta^{x^\kappa}(\overline{\tau}) = \theta_{lv}^{x^\kappa}(\overline{\tau}\ array[n])\end{array}}{\Pi;\Sigma;\Gamma;\Theta_0 \vdash_{lv} lv\underline{[}e\underline{]} : \tau\backslash\theta^{x^\kappa} \ \triangleright\ \Theta_2}$$

Select

$$\frac{\begin{array}{c}\Pi;\Sigma;\Gamma;\Theta_0 \vdash_{lv} lv : \{z_1 : \tau_1..z : \tau..z_n : \tau_n\}\backslash\theta_{lv}^{x^\kappa} \ \triangleright\ \Theta_1 \\ x = \Re(lv) \qquad \theta^{x^\kappa}(\overline{\tau}) = \theta_{lv}^{x^\kappa}(\{z_1 : \tau_1..z : \overline{\tau}..z_n : \tau_n\})\end{array}}{\Pi;\Sigma;\Gamma;\Theta_0 \vdash_{lv} lv\underline{.}z : \tau\backslash\theta^{x^\kappa} \ \triangleright\ \Theta_2}$$

Table 3.6: Type Rules for Literals.

NUM-U
$$\frac{n = len(n_1) \qquad d = len(n_2)}{\Pi; \Sigma; \Gamma; \Theta \vdash_{lit} n_1[.n_2] : \overline{num}[n.d]}$$

NUM
$$\frac{n = len(n_1) \qquad d = len(n_2)}{\Pi; \Sigma; \Gamma; \Theta \vdash_{lit} -n_1[.n_2] : \overline{num}[Sn.d]}$$

STRING-ALPHANUM
$$\frac{\{0 \,..\, 9\} \cap \texttt{"str.."} \neq \emptyset \qquad n = len(str)}{\Pi; \Sigma; \Gamma; \Theta \vdash_{lit} \texttt{"str.."} : alphanum[n]}$$

STRING-ALPHA
$$\frac{n = len(\texttt{"str.."})}{\Pi; \Sigma; \Gamma; \Theta \vdash_{lit} \texttt{"str.."} : alpha[n]}$$

TRUE
$$\frac{}{\Pi; \Sigma; \Gamma; \Theta \vdash_{lit} true : bool}$$

FALSE
$$\frac{}{\Pi; \Sigma; \Gamma; \Theta \vdash_{lit} false : bool}$$

- variable reuse involves up to 30% of overall variable usage in COBOL programs
  - nearly 90% of these, though, accumulate less than 5 storage types simultaneously within their flow-type; averagely 3
  - remaining 10% however unlikely grow wider than 8
  - 75% of non-singleton flow-types indicates reuse of numeric types
    * 80% of these come from in-place arithmetic operations possibly exceeding target variable space, such as the typical scenario `(x : num[3]) := (x : num[2]) + 1`
    * probably few of such operations are potentially risky at run-time, because programmers typically declare pictures wider than actually needed for their numerics
    * remaining 20% are re-assignments or data movements, i.e. assignments where variables on the right-hand do not appear in left-hand
- 25% of non-singleton flow-types indicates reuse of non-numeric types
  - 70% of these are alphanumeric-strings-to-array type switches and viceversa
  - 10% involve complex data types, such as nested records overlapping arrays
  - only 2% occurs between incompatible types, thus probably leading to data corruption and bugs
  - remaining 18% involve data movements implying no truncation, thus might be bad code but does not lead to run-time unwanted behaviors

Table 3.7: Type Rules for Expressions.

DEMOTE-NUM
$$\frac{\Pi;\Sigma;\Gamma;\Theta_0 \vdash_e e : num_q[\rho] \ \triangleright\ \Theta_0}{\Pi;\Sigma;\Gamma;\Theta_0 \vdash_e e : \overline{num}[\rho] \ \triangleright\ \Theta_0}$$

LV
$$\frac{\Pi;\Sigma;\Gamma;\Theta_0 \vdash_{lv} lv : \tau\backslash\theta^{x^\kappa} \ \triangleright\ \Theta_1}{\Pi;\Sigma;\Gamma;\Theta_0 \vdash_e lv : \tau \ \triangleright\ \Theta_1}$$

LIT
$$\frac{\Pi;\Sigma;\Gamma;\Theta \vdash_{lit} lit : \sigma}{\Pi;\Sigma;\Gamma;\Theta_0 \vdash_e lit : \sigma \ \triangleright\ \Theta_0}$$

NEG-S
$$\frac{\Pi;\Sigma;\Gamma;\Theta_0 \vdash_e e : \overline{num}[Sn.d] \ \triangleright\ \Theta_1}{\Pi;\Sigma;\Gamma;\Theta_0 \vdash_e -\ e : \overline{num}[Sn.d] \ \triangleright\ \Theta_1}$$

NEG-U
$$\frac{\Pi;\Sigma;\Gamma;\Theta_0 \vdash_e e : \overline{num}[n.d] \ \triangleright\ \Theta_1}{\Pi;\Sigma;\Gamma;\Theta_0 \vdash_e -\ e : \overline{num}[Sn.d] \ \triangleright\ \Theta_1}$$

NOT
$$\frac{\Pi;\Sigma;\Gamma;\Theta_0 \vdash_e e : bool \ \triangleright\ \Theta_1}{\Pi;\Sigma;\Gamma;\Theta_0 \vdash_e \underline{\texttt{not}}\ e : bool \ \triangleright\ \Theta_1}$$

PLUS-U
$$\frac{\begin{array}{c}\Pi;\Sigma;\Gamma;\Theta_0 \vdash_e e_1 : \overline{num}[n_1.d_1] \ \triangleright\ \Theta_1 \\ \Pi;\Sigma;\Gamma;\Theta_1 \vdash_e e_2 : \overline{num}[n_2.d_2] \ \triangleright\ \Theta_2 \qquad n = max(n_1,n_2) \qquad d = max(d_1,d_2)\end{array}}{\Pi;\Sigma;\Gamma;\Theta_0 \vdash_e e_1 + e_2 : \overline{num}[Sn.d] \ \triangleright\ \Theta_2}$$

PLUS-MINUS-S
$$\frac{\begin{array}{c}\Pi;\Sigma;\Gamma;\Theta_0 \vdash_e e_1 : \overline{num}[S_1n_1.d_1] \ \triangleright\ \Theta_1 \qquad \Pi;\Sigma;\Gamma;\Theta_1 \vdash_e e_2 : \overline{num}[S_2n_2.d_2] \ \triangleright\ \Theta_2 \\ S = S_1 \vee S_2 \qquad n = max(n_1,n_2)+1 \qquad d = max(d_1,d_2)\end{array}}{\Pi;\Sigma;\Gamma;\Theta_0 \vdash_e e_1(+|-)e_2 : \overline{num}[Sn.d] \ \triangleright\ \Theta_2}$$

MULT
$$\frac{\begin{array}{c}\Pi;\Sigma;\Gamma;\Theta_0 \vdash_e e_1 : \overline{num}[S_1n_1.d_1] \ \triangleright\ \Theta_1 \qquad \Pi;\Sigma;\Gamma;\Theta_1 \vdash_e e_2 : \overline{num}[S_2n_2.d_2] \ \triangleright\ \Theta_2 \\ S = S_1 \vee S_2 \qquad n = n_1 + n_2 \qquad d = d_1 + d_2\end{array}}{\Pi;\Sigma;\Gamma;\Theta_0 \vdash_e e_1 * e_2 : \overline{num}[Sn.d] \ \triangleright\ \Theta_2}$$

DIV
$$\frac{\begin{array}{c}\Pi;\Sigma;\Gamma;\Theta_0 \vdash_e e_1 : \overline{num}[S_1n_1.d_1] \ \triangleright\ \Theta_1 \qquad \Pi;\Sigma;\Gamma;\Theta_1 \vdash_e e_2 : \overline{num}[S_2n_2.d_2] \ \triangleright\ \Theta_2 \\ S = S_1 \vee S_2 \qquad n = n_1 + d_2 \qquad d = d_1 + n_2\end{array}}{\Pi;\Sigma;\Gamma;\Theta_0 \vdash_e e_1/e_2 : \overline{num}[Sn.d] \ \triangleright\ \Theta_2}$$

BIN-REL-NUM
$$\frac{\Pi;\Sigma;\Gamma;\Theta_0 \vdash_e e_1 : \overline{num}[S_1n_1.d_1] \ \triangleright\ \Theta_1 \qquad \Pi;\Sigma;\Gamma;\Theta_1 \vdash_e e_2 : \overline{num}[S_2n_2.d_2] \ \triangleright\ \Theta_2}{\Pi;\Sigma;\Gamma;\Theta_0 \vdash_e e_1 op_r e_2 : bool \ \triangleright\ \Theta_2}$$

BIN-REL-ALPHANUM
$$\frac{\Pi;\Sigma;\Gamma;\Theta_0 \vdash_e e_1 : alphanum[n1] \ \triangleright\ \Theta_1 \qquad \Pi;\Sigma;\Gamma;\Theta_1 \vdash_e e_2 : alphanum[n2] \ \triangleright\ \Theta_2}{\Pi;\Sigma;\Gamma;\Theta_0 \vdash_e e_1 op_r e_2 : bool \ \triangleright\ \Theta_2}$$

BIN-LOGIC
$$\frac{\Pi;\Sigma;\Gamma;\Theta_0 \vdash_e e_1 : bool \ \triangleright\ \Theta_1 \qquad \Pi;\Sigma;\Gamma;\Theta_1 \vdash_e e_2 : bool \ \triangleright\ \Theta_2}{\Pi;\Sigma;\Gamma;\Theta_0 \vdash_e e_1 op_l e_2 : bool \ \triangleright\ \Theta_2}$$

- 80% of jump statements require up to 3 visits (including the first one, hence 2 re-visits) to reach a convergence in the typing function status; averagely 2, hence 1 re-visit
  - 98% of those are actually pretty ordinary loops coming from COBOL iterative constructs; just 2% are weird custom cycles created by the programmer
  - remaining 20% of jump statements need anyway up to 5 visits before a convergence occurs
  - 70% the latter are actually just nested conditional loops that COBOL iterative constructs cannot express and are explicitly written by programmers via `IF` and `GOTO` statements.

All this suggests that type-flow analysis is actually able to detect a number of possible errors in COBOL programs coming from bad reuse of variables or incompatible data movements. Either ways lead to data truncation or corruption, which are the major sources of run-time bugs. And, by the way, the statistics above do not differ a lot from those collected and shown by [56].

In the following example we show how a data move from a smaller type to a larger one might lead to unwanted scenarios where previous data has not been replaced by new one:

```
{
  a := r;
  // [WARNING] reverse subsumption detected in
               assignment: right-hand type
               is smaller that left-hand type

  n := a[3];
  // [WARNING] possible access to corrupted
               data: accessing 'a' with its
               initialization type
               'alphanum[2] array[4]' but its
               content and type have changed
}
where a : alphanum[2] array[4];
      r : { x : num[3];
            y : alphanum[2];
            z : num[2] };
      n : alphanum[2]
```

Record `r` is 7-bytes long and array `a` is 8 bytes, therefore, once `r` is copied into `a`, accesses to the latter as its initialization array type would lead to unwanted data in case the last byte is accessed. Although in the example we used a literal in the

subscript, in general the analyzer cannot know what is accessed and therefore the warning is output.

For this matter, partial evaluation of expressions has been implemented in our prototype, even though we haven't considered it in this chapter[11] - that would avoid the warning in case the assignment was `n := a[1]` and we generally noted that it does slightly reduce the number of messages logged by the analyzer, overall.

3.3 Future Work

During the long research and development of this COBOL analysis system a number of side-projects have started up: some of them have been matter of thesis for Master degree, others were attempts that have not been finished but would be interesting topic of further investigation.

- A GUI front-end prototype based on Windows Presentation Foundation [21] and written in C# for .NET 4.0 has been in development until an early alpha stage. The goal was letting users browse annotated source programs and understand complex flow-type more easily. This is a purely applicative aspect of the problem, but we believe it would be crucial bringing the analyzer to a realistic production-quality level.
- Dealing with unknown statements in some interesting way, type-wise, such as adding weak types to the type-system indicating that type assumptions might get broken whenever a variable is used by a COBOL command whose semantics are unknown.
- COBOL does not have libraries but many language extensions have been put into the language standard specification over the decades. Support for the SQL extension would be useful and interesting, introducing the notion of cursor and table types within the system for detecting possible inconsistencies between declared records and actual row layout in the database.
- Adding some form of data-flow analysis over value domains and ranges. An embryonic-stage implementation of it can be already seen in the source code of the analyzer: it is based on an abstract interpretation system based on polynomials instead of usual ranges. That is of course a very complex topic on its own and we have spent some time studying it - a Master degree thesis on the subject has been written on the matter as stated on the preface.
- Designing some custom Program Understanding approaches, such as pattern recognition over identifier names or code snippets for making the system aware of typical COBOL programming trends, styles, practices and design patterns.

[11] A partial evaluation system is formalized and described in the last chapter of this thesis on Android Type Analysis.

4 Typing Android

4.1 Introduction

In this chapter we delve into another topic bordering with program validation and understanding by reconstructing types. Android is a well-known mobile platform [18] that is been having a great success among both the software industry and the open-source community.

Enhancing the Android development process is increasingly being recognized as an urgent need [14, 32, 29, 50, 27]. Surprisingly, though, there is little to no literature directly addressing the problem of typing Intents and components, nor even proposals for imposing some degree of type discipline in Android programming. In the world of development tools, analogously, there is no evidence of interest in approaching the problem from a type perspective; and all those analyzers out there deal with bytecode or install fancy runtime monitors for checking security-related properties - no one seems to care about detecting errors and issues when they are being created, i.e. when developers write code. Our proposed system represents a first step in the direction of aiding the user with a programming assistant that detects component inter-communication type issues as a second-tier type checker for Java.

Type analysis. Our proposal is in general a system for static analysis that is based on reconstructing types. We will use the term *type reconstruction* as a synonym for type inference in a wider sense: an approach based on understanding types of whatever nature by either grasping explicit type information annotated on the code or by truly inferring types from method calls, constants, variable use etc[1]. Even though there is no unification over type variables *ML* [25] in our system, this does not mean it is not a form of type reconstruction in the general sense.

Moreover, we perform some form of partial evaluation for reconstructing information belonging to the world of data in a static way - which has not to be confused with abstract interpretation, being the latter a rather different technique based on approximating values to abstract domains. In our system reconstructed types

[1] In [62] the well-known ML type inference is referred to as a form of type reconstruction.

and partially-evaluated information are then mixed into custom types representing high-level Android entities such as Intents and components. We call all this *type analysis*, because it combines static analysis techniques such as partial evaluation with type reconstruction and because the target of the analysis are the type of Android entities. In fact, once Intent and component types are collected, a number of late-time validations are performed over the program as a whole: as shown in section 4.4, inter-component communication is checked against illegal or ill-typed passing of data within Intents and consistency of requests and results is verified. In other words, our system consists of a static analysis stage that performs type reconstruction and partial evaluation, and a further validation stage that checks the types collected before and detects mismatches or inconsistencies in inter-component communication. Therefore the system is also as a type checker to some extent, rejecting Android applications that are ill-typed according to component-level type rules.

Before proceeding with the in-depth analysis of the motivational aspects of our proposal, we review the most important aspects of the Android architecture and its security model, thus providing the necessary ingredients to understand the technical contents of this chapter.

Intents. Once installed on a device, Android applications run isolated from each other in their own security sandbox. Data and functionality sharing among different applications is implemented through a message-passing paradigm built on top of *intents*, i.e., passive data structures providing an abstract description of an operation to be performed and the associated parameters. For instance, an application can send an intent to an image viewer, requesting to display a given JPEG file, to avoid the need of reimplementing such functionality from scratch.

The most interesting aspect of intents is that they can be used for both *explicit* and *implicit* communication. Explicit intents specify their intended receiver by name and are always securely delivered to it; since the identity of the recipient is typically unknown to developers of third-party applications, explicit intents are particularly useful for intra-application communication. Implicit intents, instead, do not mention any specific receiver and just require delivery to any application that supports a desired operation.

Components. Intents are delivered to application *components*, the essential building blocks of Android applications. There are four different types of components, serving different purposes:

- An *activity* represents a screen with a user interface. Activities are started with an intent and possibly return a result upon termination;

- A *service* runs in the background to perform long-running computations and does not provide a user interface. Services can either be started with an intent

or expose a remote method invocation interface to a client upon establishment of a long-standing connection;

- A *broadcast receiver* waits for intents sent to multiple applications. Broadcast receivers typically act as forwarders of system-wide broadcast messages to specific application components;

- A *content provider* manages a shared set of persistent application data. Content providers are not accessed through intents, but through a CRUD (Create-Read-Update-Delete) interface reminiscent of SQL databases.

We refer to the first three component types as "intent-based" components. Any communication among such components can employ either explicit or implicit intents.

Based on our formal type reconstruction system, we developed a prototype implementation of `Lintent`, a type-based analyzer integrated with the Android Development Tools suite (ADT). `Lintent`, among other kinds of static checks[2], features a full-fledged static analysis framework that includes Intent and component type reconstruction, partial evaluation of Java code, manifest analysis, and a suite of other actions directed towards assisting the programmer in writing more robust and reliable applications.

4.2 Motivation

The typing of Intents and component supported by the Java compiler is rather loose and uninformative: in fact, the Java type system does not keep track of any type information about either the contents of Intent objects or the data a component sends and expects to receive. This seriously hinders any form of type-based analysis and makes Android programming very error-prone. `Lintent` infers and records the types of data injected into and extracted out of intents while tracking the flow of inter-component message passing for reconstructing the incoming requests and outgoing results of each component. This proves helpful to detect common programming errors related to misuse of intents [50].

4.2.1 The problem: Untyped Intents

Often programmers write bugged code by misusing data extracted from an Intent sent by another component, even of the very same application. Intents act as untyped containers - dictionaries, more precisely - into which components put data

[2]Most notably it integrates a typing technique for privilege escalation and other security-related issue detection [11]

bound to a named label to be sent to another component; such data can be retrieved back by the recipient component from the intent as if it was a dictionary, i.e. by looking up data by specifying a label string.

This is a typical use of Intents in a type-unsafe fashion. The sender component populates the Intent object by binding data of either primitive or custom[3] type to key values.

```
class MySenderActivity extends Activity {
   static class MySer implements Serializable { ... }

   void mySenderMethod() {
      Intent i = new Intent(this, TargetActivity.class);
      i.putExtra("k1", 3);
      i.putExtra("k2", true);
      i.putExtra("k3", new MySer());
      startActivityForResult(i, 0);
   }
}
```

The recipient component retrieves data by getting the Intent and looking up data from within it by specifying the same keys. Such keys are simply objects of type `String`, therefore whether they actually are plain string literals or objects computed by complex algorithms, the Java compiler treats them in the way and their content obviously belong to the runtime world - with all the pros and cons of it. What if the recipient component looks up for an non-existing key? Or what if it retrieves a given item by specifying the wrong getter method, thus resulting in the wrong type?

```
class MyRecipientActivity extends Activity {
    static class WrongSer implements Serializable { ... }

    void onCreate(Bundle savedInstanceState) {
        Intent i = getIntent()

        // run-time type error: k1 was an int!
        String v1 = i.getStringExtra("k1");

        // dynamic cast fails!
        WrongSer o = (WrongSer)i.getSerializableExtra("k3");

        // "k2" is not even extracted: that might be unwanted!
    }
}
```

[3]By defining a `Serializable`

In short, Intents are populated by the sender by calling a family of *overloaded* putter methods `putExtra(String, τ)`, where τ belongs to a set of supported Java types; on the other end, Intents are inspected by the receiver by calling a family of getter methods `get`T`Extra(String)`, where T is a *Java type name* belonging to the set of supported types - i.e. these are *not overloaded*, but are rather a set of predefined methods with similar names.

The reason for this asymmetric design in the Intent API is due to the limitations of the Java overloading system: *context-independent* overloading does not allow the definition of multiple methods sharing the same name and arguments and differing only in the return type.

A more sophisticate form of overloading supporting *context-dependent* resolution would require a major revamp of the Java type system and would have deep implications within polymorphism and type inference[12], hence the reason why it is typically not supported by languages[4]. Nonetheless, Android API design feels quite naive in this respect: we cannot but argue whether, using Java generics[5] and other advanced language features smartly, the type of Intent contents could have been tracked by the Java type system itself somehow, offering a strong tool to the programmer for statically checking the insertion and extraction of custom data within Intents as modern standard libraries do with type-parametric container classes, for example. This poor design trend has plagued the Java community [60] since the release of generics with the 5th revision of the language, and specifically in the Android scenario it yields to a number of possible issues the receiver side:

1. key does not exist: being key just string objects and not language identifiers, they totally belong to the runtime world and are subject to runtime errors;
2. key is bound to data whose type does not match the requested type;
3. non-primitive data is retrieved simply as `Serializable` objects, which likely require a downcast to operate with, thus leading to well-known runtime type errors.

Such scenarios are not restricted to implicit Intent usage or to components communicating with unknown remote components: it may occur even between components of the same application written by the same programmer. It is exactly the same kind of error-proneness that many long-term Java programmers were used to before the release of Java 1.5: back then the language did not support generics and custom objects of any type put into containers at a given program location were

[4] Actually almost no wide-spread imperative or functional language supports context-dependent overloading resolution at all, due to its demanding type system design. Haskell supports it in form of type classes and predicates over type variables, opening the door to an advanced form of parametric polymorphism [36]. Other proposals meant for ML-like languages are just theoretical systems that have not seen the light of day in any real-world language yet [26].

[5] Android adopts Java 1.5 as language and runtime specification.

retrieved with type Object at a different program point, hence the widespread use of downcast and the resulting runtime type errors the practice implies.

4.2.2 The Proposal: Inferring Intent Types

In order to provide Android developers with some sort of tool for statically checking the consistency of Intents within their application source code, two different approaches and solutions can be proposed:

1. a replacement API that grants a more type-safe management of Intents and components, i.e a Java library wrapping a portion of the Android SDK and benefitting from generics and advanced language features for keeping track of the type of data injected into Intents by using the Java type system itself. This approach unfortunately yields to a number of drawbacks, among which the worst arguably being the inability to check existing code without having to rewrite it using the new wrapper API. Even for writing a new application from scratch - thus using a third-party API for Intents and components can be taken into consideration more realistically - the decision is tricky: developers must ensure everyone in the team mastered the new API as well as advanced Java language features, otherwise the extra safety provided by the wrapper will not pay off the extra complexity programmers have to deal with for using it properly.

2. a tool that functions as a second type-checker for the Android program - a *type analyzer* able to reconstruct the types of data injected and extracted within and from Intents. This approach does require an external program developers must be aware of and have to launch for checking their own, but it allows the user to check both existing and new code written using the standard Android API. Moreover, Android official tools for developers include a Lint[17] code analyzer that can be easily extendable by means of plug-ins. Implementing an Intent type analyzer as an ADT Lint plug-in makes it visible and widely adoptable by the dev audience, as friendly to use as an automatic and integrated second Java type-checker.

Intents are dictionaries, thus they resemble records type-wise. With a difference: labels are string objects and not language identifiers, thus they are not static entities for the type system. This makes things harder to model. In addition, Intents carry a few extra information that are tricky to reconstruct:

1. Intents can be addressed to either a recipient component class (explicit Intent) or an action string (implicit Intent). Different constructors of the Intent class are defined by the Android framework for creating explicit or implicit Intents - but due to the typical imperative programming conventions, empty constructor and setter methods can be used instead for configuring up Intents. This leads

to the need of performing a full-featured form of analysis for reconstructing the kind of Intents reliably.

2. components can invoke method `Activity.startActivityForResult(Intent, int)` for sending Intents and binding them to an integer *request code* to be used by the receiver for discriminating Intents sent by the same sender component for different purposes. Request codes appear multiple times in other spots of a component architecture: once a recipient activity performs its task, an Intent can be sent back to the original sender along with a special result code and the request code that was specified on the `startActivityForResult()`. This information can be retrieved by analyzing the sender component callback method `Activity.onActivityResult(Intent, int, int)`.

What makes Intent type reconstruction challenging is basically a subtle aspect of the Android API: strings used as Intent keys or action strings for implicit communication, result and request codes, as well as Intent objects themselves passed as argument to component methods are all runtime data: these values must be somehow evaluated statically for being used in any type-based approach - and this makes things much more harder than most type-based analysis or reconstruction techniques, as it involves some form of data-flow analysis within the type reconstruction process.

4.2.3 Challenges

Analyzing Android applications actually means analyzing Java code deeply based on a special API that introduces a number of anomalies with respect to canonical Java programming patterns and conventions.

As an example, the control-flow of Android applications is not the usual one: program does not start up from the `main` method and does not proceed linearly. Activity and Service components are invoked by the operating system by calling the `onCreate()` method - developers must know this and design their app according to the application life-cycle [20]. No entry point exists for the application alone but rather one entry point exists for each component, making the analysis in need of being specifically designed for Android. In other words, within method bodies the flow is that of plain Java, while at top level the call flow is handled by Android in a special way.

Many challenges have to be faced for offering a proper analysis.

Detecting API patterns. The Android communication API offers various different patterns to implement inter-component communication - all of which are used by the development community, possibly even mixed in the same application. For example, the developer guide describes at least three different ways to implement bound services, with different degrees of complexity, and a local inspection of the

instructions alone does not suffice to reconstruct enough information to support verification. Partial evaluation techniques combined with type reconstruction are needed where syntactic pattern matching of code templates would be too naive.

Delocalized information. Request codes specified by calls to `startActivityForResult()` and passed to `onActivityResult()` by the framework are defined and used in different locations within the application code. Component names and certain information related to them (such as permissions) are meta-information which does not appear in the Java code, but in the application Manifest file. This is an XML file containing, among other information, the permissions each application component requires for being accessed and what permissions are requested by the application itself. Information is therefore spread across different files or spots within files.

Type reconstruction. Arguably the hardest challenge is related to a number of "untyped" programming conventions which are enabled by the current Android API. In section 4.2.1 the example highlights a total lack of static control over standard intents manipulation operations: with these premises, no type-based analysis can be soundly performed. For this reason, intents are basically treated as record types of the form $\{k_1 : T_1, \ldots, k_n : T_n\}$, where k_i is a string constant and T_i is a Java type. This enforces a much stronger discipline on data passing between components - i.e., on the injection and extraction of "extras" into and from intents. Notably, the same type reconstruction applies to objects of type `Bundle` as well, and `Bundle` objects possibly put within Intents or other `Bundle`'s are recursively typed as sub-records.

Partial evaluation. Recall from the previous discussion that every data an user puts into an intent must be bound to a key, hence an Intent could be initially represented as a record of the form $\{k_1 : \tau_1 \ldots k_n : \tau_n\}$, where τ_i are Java types for $1 <= i <= n$. Unfortunately, the dictionary keys k_i are runtime string objects computed by whatever Java expression of type string - they are not first-class language identifiers, thus they are not easily known at compile-time. In other words, whether they happen to appear as string literals or complex method calls computing a string object is irrelevant: in any case they belong to the run-time world. The very same problem arises for result codes, request codes, implicit Intent action strings and explicit Intent recipient component class objects: all this information could be the result of computations. In order to reconstruct these bits, some form of static analysis should be endeavored. We argue that *partial evaluation* is enough: a good trade off between complexity and results, as typically Android applications to not rely on sophisticate algorithms for computing Intent keys, recipients and result codes, but they often calculate them by means of operations on constants or simple function calls - defining `static final` attributes as constants is a pretty common practice in Android development. In the real-world a partial evaluator would actually work most of the times and is more straightforward to formalize and integrate into a type

system.

Consider the following code snippet reproducing a typical way of organizing Intent information in an Android program:

```
static class Const {
    public static final label = "LABEL";
    public static final name = name + "_NAME";
    public static final age = name + "_AGE";
    public static final code = 4;
}

Class<?> getRecp() { return SomeClass.class; }

Activity getSelf() { return this; }

void mySenderMethod() {
    int code_base = 10;
    final String key_base = "PERSON_" + Const.label;
    Intent i = new Intent(getSelf(), getRecp());
    i.putExtra(key_base + Const.name + 1, "John");
    i.putExtra(key_base + Const.name + 2, "Smith");
    i.putExtra(key_base + Const.age, 23);
    startActivityForResult(i, code_base + Const.code);
}
```

Here the system should be able to reconstruct something more than just a record: besides calculating key string values by partially evaluating the first argument of each `putExtra()` call, the recipient class object `SomeClass.class` and the request code `3` must be added to the Intent representation. That requires a rich record of form $(|T_c|^R, x)\{k_1 : T_1..k_n : T_n\}$, where $|T_c|$ is the class object of the reifiable [6] component type T_c, and x is an integer constant.

Interaction with third party libraries. Typically applications rely on external libraries offering a number of services to the programmer. From the point of view of Java code, such libraries are collections of compiled classes linked into one or more `jar` files: their source code is therefore not available at analysis time. Import declarations on top of compilation units simply carry information on package names and class paths, but do not specify class member signatures or other details. This is mainly an implementation challenge, but type resolution is a tricky task for a tool that does not have the same information the Java compiler is given by command line arguments, therefore types that are inferred as external must be treated in some special way: access to `jar` files must be granted to `Lintent` to let it inspect the contents of imported packages and classes.

[6] Since some type information is erased during compilation, in Java not all types are available at run time. Types that are completely available at run-time are known as reifiable types [22].

4.3 Type System

We defined a system built on top of Java type system[22] for modeling the types of Intents and of components. In table 4.1 we show the Java AST of expressions, statements, literals and types. Only constructs meaningful to our formalization are shown; those omitted, though, would have added little to nothing to our reasoning.

In table 4.2 we show the type of Intents and the values computed by partial evaluation used by our system for performing its reconstruction.

Notice that literal constants of form $\ell[l]$ do not carry along any type information because literals of different primitive types are syntactically distinct, therefore syntax-directed rules can match accordingly. Constants can also be pairs $\omega : \tau$ that represents pointers to heap objects of type τ. Addresses can either be unknown ($\circ$) or actual references, which consist simply of strictly-positive integer numbers r identifying an object instance uniquely.

The type of Intents are basically records binding keys consisting in string constants to types, plus a recipient which can either be undefined (for example when an Intent object is created with the empty constructor), explicit (a specific component class name) or implicit (a string constant representing an Android action string).

Finally, the type of components are made of multiple sets of references to Intents r or of pairs reference-code (r, c). Activity components have form $\mathcal{I}_1 \rightarrow \mathcal{I}_2 \Longrightarrow \mathcal{L}_1 \hookleftarrow \mathcal{L}_2$ and each part of the type representation models a direction of the inter-component communication system on which Android is based:

- *Incoming Requests* $\mathcal{I}_1$. In the `onCreate()` method an Activity can use `getIntent()` for retrieving an Intent sent by another sender component. Such Intents can possibly be multiple as our system creates a fresh Intent representation for each `getIntent()`. Considering Activities as if they were functions, this would be the input, i.e. the domain of the function, hence the Intent set $\mathcal{I}_1$ appearing in the Activity type.

- *Outgoing Results* $\mathcal{I}_2$. After having processes an incoming Intent, an Activity can use the `setResult()` methods for repling to the original sender component sending back an Intent - and possibly attaching a request code to it. This is similar as the codomain of the Activity function.

- *Outgoing Requests* $\mathcal{L}_1$. An Activity can act both as a receiver and as a sender at the same time: the functional-like arrow type is not enough for modeling an Activity properly. Outgoing requests can be Intents possibly bound to request codes and sent via the `startActivity()` and `startActivityForResult()` methods.

- *Incoming Results* $\mathcal{L}_2$. The last doorway of an Activity is the `onActivityResult()` method callback, which is invoked by the OS when an Activity receives a result to an outgoing request sent via `startActivityForResult()`. Multiple Intents

Table 4.1: Abstract Syntax of Java Types, Expressions and Statements..

τ	$:=$		**type**
		$\top$	**top type**
	$\mid$	$\bot$	**bottom type**
	$\mid$	`int` $\mid$ `boolean` $\mid$ ..	**built-in primitive types**
	$\mid$	`T`	**fully-qualified class name**
	$\mid$	$\tau[]$	**array**
	$\mid$	$\tau < \tau_1,\ ..,\ \tau_n >$	**application**
st	$:=$		**statement**
		$\tau\ x^{[F]} = e$	**declaration with initializer**
	$\mid$	e	**statement expression**
	$\mid$	`if` e_1 `then` st_1 `else` st_2	**if-then-else**
	$\mid$	st_1 `;` st_2	**sequence**
	$\mid$	x `:=` e	**assignment**
	$\mid$	`return` e	**return**
	$\mid$	`while` e `do` st	**while**
e	$:=$		**expression**
		l	**literal**
	$\mid$	x	**var**
	$\mid$	`this`	**this**
	$\mid$	$e.x$	**select**
	$\mid$	$e.m(e_1, .., e_n)$	**call**
	$\mid$	`new` $\tau(e_1, .., e_n)$	**construction**
	$\mid$	`new` τ`[]`$\{e_1, .., e_n\}$	**array construction**
	$\mid$	e_1 `?` e_2 `:` e_3	**conditional**
	$\mid$	$(\tau)\ e$	**type cast**
	$\mid$	$e_1\ binop\ e_3$	**binary operator**
	$\mid$	$unop\ e$	**unary operator**
	$\mid$	e_1`[`e_2`]`	**array subscript**
l	$:=$		**literal**
		n	**integer**
	$\mid$	d	**double-precision float**
	$\mid$	`true` $\mid$ `false`	**boolean**
	$\mid$	s	**string**
	$\mid$	$[\![\upsilon_1, .., \upsilon_n]\!]$	**array**
	$\mid$	`null`	**null**
	$\mid$	`T.class`	**class operator**

`where T is a fully-qualified Java class identifier,` x `is an identifier and` F `is t`

Table 4.2: Syntax of Types for Intent and Components..

υ	$:=$		**partially evaluated constant**
		ϵ	**none**
	$\mid$	$\ell[l]$	**literal**
	$\mid$	$\omega : \tau$	**address of heap object**
ω	$:=$		**object address**
		$\circ$	**unknown object**
	$\mid$	$r \mapsto \circ$	**reference to object**
$\imath$	$:=$	$(\varrho)\{\ s_1 : \tau_1 \ ..\ s_n : \tau_n\ \}$	**intent type**
ϱ	$:=$		**intent recipient**
		`T.class`	**explicit component class**
	$\mid$	`@`s	**implicit action string**
	$\mid$	`?`	**implicit action string**
ξ	$:=$		**component type**
		$\mathcal{I}_1 \rightarrow \mathcal{I}_2 \Longrightarrow \mathcal{L}_1 \hookleftarrow \mathcal{L}_2$	**activity**
	$\mid$	$\mathcal{I}_1 \Rightarrow \mathcal{I}_2$	**service**
$\mathcal{I}$	$:=$	$\{\ r_1 \ ..\ r_n\ \}$	**intent reference set**
$\mathcal{L}$	$:=$	$\{\ (r_1, c_1) \ ..\ (r_n, c_n)\ \}$	**intent and request code set**
c	$:=$		**request code**
		n	**some code**
		$\cdot$	**none**

`where T is a fully-qualified Java class identifier, all` s `are strings, all` $n \in \mathbb{N}$ `an`

might be reconstructed by our system and each can be bound to a different request codes.

Service components are simpler: they just consist in the incoming request and the outgoing request parts, being both Intent sets only.

As a final note, request codes do not have to be mistaken for result codes: the former must be taken into account by our system for associating requests to replies, while the latter do not appear in our model because they are just integers the Android API uses for specifying success or failure by a given Activity - similarly to what happens in most operating systems with process exit codes.

4.3.1 Formalization

Before giving rules for partial evaluation and type reconstruction, we first make a number of definitions.

As defined by the Java specification [22], a reifiable type is a type that is available at runtime - that is after type erasure has been performed by the type checker. Therefore, types can be translated into reifiable types by means of a simple type-to-type transformation that reproduces the behaviour of Java type erasure, i.e. basically removing arguments from type applications and substituting primitive types with their object-based counterparts.

Definition 4.3.1 (Reification)**.** We define a reification function $\lceil \tau \rceil : \tau \to \tau$ mapping Java types to reifiable types.

$$
\begin{array}{rcl}
\lceil \texttt{int} \rceil & = & \texttt{java.lang.Integer} \\
\lceil \texttt{boolean} \rceil & = & \texttt{java.lang.Boolean} \\
 & \vdots & \\
\lceil \texttt{T} \rceil & = & \texttt{T} \\
\lceil \tau[] \rceil & = & \lceil \tau \rceil [] \\
\lceil \tau < \tau_1, .., \tau_n > \rceil & = & \lceil \tau \rceil
\end{array}
$$

Definition 4.3.2 (Subtyping)**.** We define a simple subtype relation between Java types such that $\tau_1 \sqsubseteq \tau_2$ iff $\tau_1 = \tau_2$ or T_1 extends a type τ_3 such that $\tau_3 \sqsubseteq \tau_2$, where $T_1 = \lceil \tau_1 \rceil$ and $T_2 = \lceil \tau_2 \rceil$. We also define the top type $\top$ as the type such that, for all types τ, the relation $\tau \sqsubset \top$ holds; and the bottom type such that $\sqsubset \tau$ holds.

Note that the bottom type is used just for typing the `null` literal, precisely as the Java type compiler does.

For inspecting the content of classes we need to define them as set of members, in such a way that, for example, $(a : \tau) \in \texttt{T}$ holds for an attribute identifier a and a Java class name `T`.

Definition 4.3.3 (Class as Set of Members)**.** Let τ be a Java type and $\mathtt{T} = \lceil\tau\rceil$ its reifiable type representing a class. We treat classes as sets of attributes, inner classes and methods (constructors are treated as static methods whose return type is the self type). Members are bindings of form:

$$
\begin{array}{rcll}
a^{[F]} & : & \tau & \texttt{attribute} \\
a^{[F]} & : & \tau = e & \texttt{attribute with initializer} \\
\mathtt{T} & : & \tau & \texttt{inner class} \\
m & : & x_1 : \tau_1 \times .. \times x_n : \tau_n \to \tau_0 = st & \texttt{method with body}
\end{array}
$$

Flag F denotes the optional final modifier.

Mind that in Java attributes and inner classes cannot be overloaded (thus bindings for these are unique), while methods are subject to a context-independent form of overloading - which means that multiple bindings with the same method name might occur, thus lookup must take into account the type of arguments of a given method call. Hence, the need for two distinct lookup operators. The most basic one looks up non-overloaded members (i.e. attributes and inner classes) within Java classes. $\tau \downarrow x$ seeks for symbol x within the reified type $\lceil\tau\rceil$ and returns the type associated to member x; it recurs on super classes in case none is found until the top type is reached.

Definition 4.3.4 (Lookup Member)**.** Let $\tau \downarrow x$ be the lookup member function defined as follows:

$$
\tau \downarrow x = \begin{cases} \epsilon & \tau = \top \\ \tau_x & (x : \tau_x) \in \lceil\tau\rceil \\ \sigma \downarrow x & x \notin \lceil\tau\rceil \wedge \tau \sqsubset \sigma \end{cases}
$$

Where σ is the direct super type of τ.

The second member lookup operator is for resolving overloaded methods: $\tau \Downarrow m(\tau_1, .., \tau_n)$ seeks for possibly multiple methods having name m within the reified type $\lceil\tau\rceil$ and calculates the best match by minimum distance between types of arguments and parameters.

Definition 4.3.5 (Type Distance)**.** Given two Java types τ_1 and τ_2, the type distance $\tau_1 \bowtie \tau_2$ is defined recursively as follows:

$$
\tau_1 \bowtie \tau_2 = \begin{cases} 0 & \tau_1 = \tau_2 \\ \infty & \tau_1 = \top \\ 1 + \sigma_1 \bowtie \tau_2 & \tau_1 \sqsubset \sigma_1 \end{cases}
$$

Where σ_1 is the direct super type of τ_1.

Definition 4.3.6 (Resolve Overload)**.** Let $\tau \Downarrow m(\tau_1, .., \tau_n)$ be the resolution function calculating the result $(m : x_1 : \overline{\tau_1} \times .. \times x_n : \overline{\tau_n} \rightarrow \overline{\tau_0} = st)$ such that $\nexists(m : x'_1 : \tau'_1 \times .. \times x'_n : \tau'_n \rightarrow \tau'_0 = st') \in \mathtt{T} \mid \sum_{i=1}^{n} \tau_i \bowtie \tau'_i < \sum_{i=1}^{n} \tau_i \bowtie \overline{\tau_i}$, where $\mathtt{T} = \tau \downarrow m$.

Finally, a number of formal tools must be introduced for dealing with literals and constants. Literals are language elements that can't be manipulated formally: they need to be converted into mathematical constants and objects in such a way that partial evaluation rules can handle them properly. For example, the literal `3` does not belong to natural or integer numbers: it is just a piece of code, but it can easily be mapped to integer For the sake of simplicity we mention only integers, double-precision floats, strings and booleans.

Definition 4.3.7 (Literal Sets)**.** We define $Ł_\tau$ as the set of literals of a primitive type τ. And the set of all Java literals $Ł = Ł_{\mathtt{int}} \cup Ł_{\mathtt{boolean}} \cup Ł_{\mathtt{double}} \cup Ł_{\mathtt{String}}$.

Definition 4.3.8 (String Set and Append)**.** We define $\mathbb{S}$ as the set of all character strings that can be specified by the string literals contained in $Ł_{\mathtt{String}}$. And the string append binary operator $\uplus : \mathbb{S} \times \mathbb{S} \rightarrow \mathbb{S}$.

Definition 4.3.9 (Invertible Literal Abstraction)**.** We define a family of injective functions for mapping Java literals to numeric, boolean and string constants in the mathematical sets.

$$\alpha_n : Ł_{\mathtt{int}} \rightarrow \mathbb{Z} \qquad \alpha_n(\ell[n]) = \begin{cases} \vdots & \\ 0 & n \equiv \mathtt{0} \in Ł_{\mathtt{int}} \\ 1 & n \equiv \mathtt{1} \in Ł_{\mathtt{int}} \\ \vdots & \end{cases}$$

$$\alpha_d : Ł_{\mathtt{double}} \rightarrow \mathbb{R} \qquad \alpha_d(\ell[d]) = \begin{cases} \vdots & \\ 0.0 & d \equiv \mathtt{0.0} \in Ł_{\mathtt{double}} \\ \vdots & \\ 1.0 & d \equiv \mathtt{1.0} \in Ł_{\mathtt{double}} \\ \vdots & \end{cases}$$

$$\alpha_s : Ł_{\mathtt{String}} \rightarrow \mathbb{S} \qquad \alpha_s(\ell[s]) = \begin{cases} "blabla" & s \equiv \mathtt{"blabla"} \in Ł_{\mathtt{String}} \\ \vdots & \end{cases}$$

$$\alpha_b : Ł_{\mathtt{boolean}} \rightarrow \{true, false\} \qquad \alpha_b(\ell[b]) = \begin{cases} true & b \equiv \mathtt{true} \in Ł_{\mathtt{boolean}} \\ false & b \equiv \mathtt{false} \in Ł_{\mathtt{boolean}} \end{cases}$$

Being clearly injective, an inverse function α^{-1} mapping constants back to Java literals exists for each of the functions above.

4.3.2 Environments and Judices

We define the following environments appearing in the premise of typing and partial evaluation rules. For the sake of simplicity we consider all Java types as fully qualified, as it would otherwise complicate things due to the need to resolve type paths within packages, nested classes and external libraries - a challenging task that adds nothing to the principles of our system (but that has been necessarily implemented in `Lintent`). Therefore any occurrence of τ means a fully qualified type and `T` a fully qualified reifiable type.

Definition 4.3.10 (Variable Type Environment Γ)**.** We define the type environment Γ as a map from identifiers to types and partial evaluated constants. Bindings have form $x : \tau \rightsquigarrow \nu$. We may either say $x \in \Gamma$ or $(x : \tau) \in \Gamma$ or $(x : \tau \rightsquigarrow \nu) \in \Gamma$ for denoting any binding belonging to Γ possibly ignoring type and constant information.

Definition 4.3.11 (Intent Environment Ω)**.** We define an auxiliary environment Ω as a map from object references to intent types. Bindings have form $r \mapsto \imath$ where $r \in \mathbb{N}^*$.

Definition 4.3.12 (Component Environment Ξ)**.** We define another auxiliary environment Ξ as a map from component names (i.e. reifiable Java type) to component types. Bindings have form $T : \xi$.

Rules are given for two kinds of judices: rules for Partial Evaluation and rules for Type Reconstruction. The two kinds are inter-connected and therefore all environments involved appear in the premise as well as in the effect side of both.

- *Partial Evaluation.* Partial evaluation judices are meant to evaluate language terms to constants υ. Environments Ω and Ξ are involved but are never touched: they're just passed to type reconstruction rules. Nonetheless, the whole system must propagate all environments thoroughly.
  - *Expressions.* Rules of form $\widetilde{\tau}; \Gamma; \Omega; \Xi \vdash e \rightsquigarrow \upsilon \rhd \Gamma; \Omega; \Xi$ means that, in the given environments, while typing class members whose type of `this` is $\widetilde{\tau}$, expression e evaluates to constant υ and environments in the premise are effected.
  - *Statements.* Rules of form $\widetilde{\tau}; \Gamma; \Omega; \Xi \vdash st \rhd \Gamma'; \upsilon$ means that, in the given environment, while typing class members whose type of `this` is $\widetilde{\tau}$, statement st returns constant υ and the environments in the premise are effected.
- *Typing.* Typing judices basically give a type to terms - similarly to the judices for an ordinary type system. For reconstructing the type of Intents and components environments Γ, Ω and Ξ are involved.

- *Expressions.* Rules of form $\widetilde{\tau}; \Gamma; \Omega; \Xi \vdash e : \tau \rhd \Gamma'; \Omega'; \Xi'$ means that, in the given environments, while typing class members whose type of `this` is $\widetilde{\tau}$, expression e can be given the type τ and environments in the premise are effected.
- *Statements.* Rules of form $\widetilde{\tau}; \Gamma; \Omega; \Xi \vdash st \rhd \Gamma'; \upsilon$ means that, in the given environments, while typing class members whose type of `this` is $\widetilde{\tau}$, statement *st* can be typed successfully and the environments in the premise are effected.

4.3.3 Partial Evaluation Rules

In general, partial evaluation rules are used by type reconstruction rules shown below: the opposite happens only in the CALL rule which has to know the type of arguments for resolving an overloaded method that has to be evaluated. Also, all rules resemble a continuation-passing style (CPS) [8]: environments in the thesis premise are inputs and those on the right hand of the $\rhd$ symbol are outputs. Environments are passed through all rules in the hypothesis by forwarding the output of a rule as the input of the following one.

Keep in mind that our system performs a type analysis on Java source code that *has already been compiled*. This means that we can skip several generic type checking rules, having the code been previously validated by the Java compiler: for example, when reconstructing the type of an array constructor invocation, we do not need to check that all expressions in the initializer have a subtype of the type specified for the array; our system can proceed with evaluation of all expressions assuming type errors cannot occur.

Evaluating Expressions. Partial evaluation rules for expressions are given in table 4.3. Mind also that in table 4.4 we won't show rules for all binary and unary operators, as they would be all similar. We now propose a detailed description of the most meaningful ones.

(FALLBACK) Rules must be intended to work when all hypotheses apply; whenever they don't, rule (FALLBACK) holds by matching any expression and evaluates ϵ, meaning that an expression could not be evaluated statically.

(BIN-*op*) We chose a few meaningful representatives for binary and unary operators, since other cases are analogous. The basic idea is that when operands evaluate successfully to constant literals, the literal abstraction function (as of definition 4.3.9) is applied to such literals for mapping them into the ordinary mathematical sets ($\mathbb{Z}, \mathbb{R}$, ecc.) in order to compute a result statically by means of the appropriate arithmetic operator. Result is eventually mapped back to literal using the inverse function of literal abstraction.

Table 4.3: Partial Evaluation Rules for Java Expressions..

Lit

$$\frac{}{\widetilde{\tau}; \Gamma; \Omega; \Xi \vdash l \rightsquigarrow \ell[l] \rhd \Gamma; \Omega; \Xi}$$

This

$$\frac{}{\widetilde{\tau}; \Gamma; \Omega; \Xi \vdash \texttt{this} \rightsquigarrow \circ : \widetilde{\tau} \rhd \Gamma; \Omega; \Xi}$$

Var-Local

$$\frac{(x : \tau \rightsquigarrow \upsilon) \in \Gamma \vee (x^F : \tau \rightsquigarrow \upsilon) \in \Gamma}{\widetilde{\tau}; \Gamma; \Omega; \Xi \vdash x \rightsquigarrow \upsilon \rhd \Gamma; \Omega; \Xi}$$

Var-Member

$$\frac{\widetilde{\tau}; \Gamma; \Omega; \Xi \vdash \texttt{this}.x \rightsquigarrow \upsilon \rhd \Gamma; \Omega; \Xi}{\widetilde{\tau}; \Gamma; \Omega; \Xi \vdash x \rightsquigarrow \upsilon \rhd \Gamma; \Omega; \Xi}$$

Sel

$$\frac{\widetilde{\tau}; \Gamma_0; \Omega_0; \Xi_0 \vdash e \rightsquigarrow \omega : \tau \rhd \Gamma_1; \Omega_1; \Xi_1 \qquad \tau \downarrow x = (x^F : \tau_1 = \upsilon_1)}{\widetilde{\tau}; \Gamma_0; \Omega_0; \Xi_0 \vdash e.x \rightsquigarrow \upsilon_1 \rhd \Gamma_1; \Omega_1; \Xi_1}$$

New

$$\frac{\begin{array}{c}\widetilde{\tau}; \Gamma_{i-1}; \Omega_{i-1}; \Xi_{i-1} \vdash e_i : \tau_i \rhd \Gamma_i; \Omega_i; \Xi_i \ (\forall i \in [1, n]) \\ \texttt{T} = \lceil \tau \rceil \qquad \tau \Downarrow \texttt{T} = (\texttt{T} : x_1 : \tau_1 \times .. \times x_n : \tau_n \to \tau = st) \qquad r \in \mathbb{N}^* \ \texttt{fresh}\end{array}}{\widetilde{\tau}; \Gamma_0; \Omega_0; \Xi_0 \vdash \texttt{new}\ \tau(e_1, .., e_n) \rightsquigarrow r \mapsto \circ : \tau \rhd \Gamma_n; \Omega_n; \Xi_n}$$

Cond-True

$$\frac{\widetilde{\tau}; \Gamma_0; \Omega_0; \Xi_0 \vdash e_1 \rightsquigarrow \ell[true] \rhd \Gamma_1; \Omega_1; \Xi_1 \qquad \widetilde{\tau}; \Gamma_1; \Omega_1; \Xi_1 \vdash e_2 \rightsquigarrow \upsilon \rhd \Gamma_2; \Omega_2; \Xi_2}{\widetilde{\tau}; \Gamma_0; \Omega_0; \Xi_0 \vdash e_1 \ \texttt{?}\ e_2 \ \texttt{:}\ e_3 \rightsquigarrow \upsilon \rhd \Gamma_2; \Omega_2; \Xi_2}$$

Cond-False

$$\frac{\widetilde{\tau}; \Gamma_0; \Omega_0; \Xi_0 \vdash e_1 \rightsquigarrow \ell[false] \rhd \Gamma_1; \Omega_1; \Xi_1 \qquad \widetilde{\tau}; \Gamma_1; \Omega_1; \Xi_1 \vdash e_3 \rightsquigarrow \upsilon \rhd \Gamma_2; \Omega_2; \Xi_2}{\widetilde{\tau}; \Gamma_0; \Omega_0; \Xi_0 \vdash e_1 \ \texttt{?}\ e_2 \ \texttt{:}\ e_3 \rightsquigarrow \upsilon \rhd \Gamma_2; \Omega_2; \Xi_2}$$

Call

$$\frac{\begin{array}{c}\widetilde{\tau}; \Gamma_i; \Omega_i; \Xi_i \vdash e_i : \tau_i \rhd \Gamma_{i+1}; \Omega_{i+1}; \Xi_{i+1} (\forall i \in [0, n]) \\ \tau_0 \Downarrow m(\tau_1, .., \tau_n) = (m : x_1 : \overline{\tau_1} \times .. \times x_n : \overline{\tau_n} \to \overline{\tau} = \overline{st}) \\ \tau_i \sqsubseteq \overline{\tau_i} \ (\forall i \in [1, n]) \qquad \widetilde{\tau}; \Gamma; \Omega; \Xi \vdash e_i \rightsquigarrow \upsilon_i \rhd \ (\forall i \in [0, n]) \\ \widetilde{\tau}; \Gamma_{n+1}, x_1 : \overline{\tau_1} \rightsquigarrow \upsilon_1..x_n : \overline{\tau_n} \rightsquigarrow \upsilon_n \vdash \overline{st} \rhd \overline{\Gamma}; \overline{\upsilon}\end{array}}{\widetilde{\tau}; \Gamma_0; \Omega_0; \Xi_0 \vdash e_0.m(e_1, .., e_n) \rightsquigarrow \overline{\upsilon} \rhd \Gamma_{n+1}; \Omega_{n+1}; \Xi_{n+1}}$$

Cast

$$\frac{\widetilde{\tau}; \Gamma_0; \Omega_0; \Xi_0 \vdash e \rightsquigarrow \omega : \tau_e \rhd \Gamma_1; \Omega_1; \Xi_1}{\widetilde{\tau}; \Gamma_0; \Omega_0; \Xi_0 \vdash (\tau)\ e \rightsquigarrow \omega : \tau \rhd \Gamma_1; \Omega_1; \Xi_1}$$

Subscript

$$\frac{\begin{array}{c}\widetilde{\tau}; \Gamma_0; \Omega_0; \Xi_0 \vdash e_1 \rightsquigarrow \ell[[\![\overline{\upsilon}_1, .., \overline{\upsilon}_m]\!]] \rhd \Gamma_1; \Omega_1; \Xi_1 \\ \widetilde{\tau}; \Gamma_1; \Omega_1; \Xi_1 \vdash e_2 \rightsquigarrow \ell[n] \rhd \Gamma_2; \Omega_2; \Xi_2 \qquad (0 \leq n \leq m-1)\end{array}}{\widetilde{\tau}; \Gamma_0; \Omega_0; \Xi_0 \vdash e_1 \texttt{[} e_2 \texttt{]} \rightsquigarrow \overline{\upsilon}_{n+1} \rhd \Gamma_2; \Omega_2; \Xi_2}$$

Fallback

$$\frac{}{\widetilde{\tau}; \Gamma; \Omega; \Xi \vdash e \rightsquigarrow \epsilon \rhd \Gamma; \Omega; \Xi}$$

Table 4.4: Partial Evaluation Rule Samples for Java Operator Expressions and Literals..

UN-NEG-DOUBLE

$$\frac{\widetilde{\tau};\Gamma_0;\Omega_0;\Xi_0 \vdash e_1 \rightsquigarrow \ell[d_1] \rhd \Gamma_1;\Omega_1;\Xi_1 \qquad d_2 = \alpha_d^{-1}(0.0 - \alpha_d(d_1)) \qquad d_2 \in \L_{\texttt{double}}}{\widetilde{\tau};\Gamma_0;\Omega_0;\Xi_0 \vdash \texttt{-}e_1 \rightsquigarrow \ell[d_2] \rhd \Gamma_1;\Omega_1;\Xi_1}$$

UN-POSTINC-INT

$$\frac{\widetilde{\tau};\Gamma_0;\Omega_0;\Xi_0 \vdash x \rightsquigarrow \ell[n_1] \rhd \Gamma_1;\Omega_1;\Xi_1 \qquad n_2 = \alpha_n^{-1}(\alpha_n(n_1)+1) \qquad n_2 \in \L_{\texttt{double}}}{\widetilde{\tau};\Gamma_0;\Omega_0;\Xi_0 \vdash x\texttt{++} \rightsquigarrow \ell[n_1] \rhd \Gamma_1,(x:\texttt{int} \rightsquigarrow \ell[n_2]);\Omega_1;\Xi_1}$$

BIN-PLUS-INT

$$\frac{\begin{array}{c}\widetilde{\tau};\Gamma_0;\Omega_0;\Xi_0 \vdash e_1 \rightsquigarrow \ell[n_1] \rhd \Gamma_1;\Omega_1;\Xi_1 \qquad \widetilde{\tau};\Gamma_1;\Omega_1;\Xi_1 \vdash e_2 \rightsquigarrow \ell[n_2] \rhd \Gamma_2;\Omega_2;\Xi_2 \\ n_3 = \alpha_n^{-1}(\alpha_n(n_1)+\alpha_n(n_2)) \qquad n_1,n_2,n_3 \in \L_{\texttt{int}}\end{array}}{\widetilde{\tau};\Gamma_0;\Omega_0;\Xi_0 \vdash e_1\texttt{+}\ e_2 \rightsquigarrow \ell[n_3] \rhd \Gamma_2;\Omega_2;\Xi_2}$$

BIN-PLUS-STRING

$$\frac{\begin{array}{c}\widetilde{\tau};\Gamma_0;\Omega_0;\Xi_0 \vdash e_1 \rightsquigarrow \ell[s_1] \rhd \Gamma_1;\Omega_1;\Xi_1 \qquad \widetilde{\tau};\Gamma_1;\Omega_1;\Xi_1 \vdash e_2 \rightsquigarrow \ell[s_2] \rhd \Gamma_2;\Omega_2;\Xi_2 \\ s_3 = \alpha_s^{-1}(\alpha_s(s_1) \uplus \alpha_s(s_2)) \qquad s_1,s_2,s_3 \in \L_{\texttt{String}}\end{array}}{\widetilde{\tau};\Gamma_0;\Omega_0;\Xi_0 \vdash e_1\texttt{+}\ e_2 \rightsquigarrow \ell[s_3] \rhd \Gamma_2;\Omega_2;\Xi_2}$$

BIN-REL-LT

$$\frac{\begin{array}{c}\widetilde{\tau};\Gamma_0;\Omega_0;\Xi_0 \vdash e_1 \rightsquigarrow \ell[n_1] \rhd \Gamma_1;\Omega_1;\Xi_1 \qquad \widetilde{\tau};\Gamma_1;\Omega_1;\Xi_1 \vdash e_2 \rightsquigarrow \ell[n_2] \rhd \Gamma_2;\Omega_2;\Xi_2 \\ b = \alpha_b^{-1}(\alpha_n(n_1) < \alpha_n(n_2)) \qquad n_1,n_2 \in \L_{\texttt{int}} \qquad b \in \L_{\texttt{boolean}}\end{array}}{\widetilde{\tau};\Gamma_0;\Omega_0;\Xi_0 \vdash e_1\texttt{<}\ e_2 \rightsquigarrow \ell[b] \rhd \Gamma_2;\Omega_2;\Xi_2}$$

LIT-INT

$$\frac{}{\widetilde{\tau};\Gamma;\Omega;\Xi \vdash n \rightsquigarrow \ell[n] \rhd \Gamma;\Omega;\Xi}$$

NEW-ARRAY

$$\frac{\widetilde{\tau};\Gamma_{i-1};\Omega_{i-1};\Xi_{i-1} \vdash e_i \rightsquigarrow \upsilon_i \rhd \Gamma_i;\Omega_i;\Xi_i \ (\forall i \in [1,n])}{\widetilde{\tau};\Gamma_0;\Omega_0;\Xi_0 \vdash \texttt{new}\ \tau\texttt{[]}\ \{e_1,..,e_n\} \rightsquigarrow \ell[[\![\upsilon_1,..,\upsilon_n]\!]] \rhd \Gamma_n;\Omega_n;\Xi_n}$$

(Un-PostInc-Int) The semantics of post-increment in Java implies a special side-effected rule that, rather than behaving like other *pure* operators, it resembles (Assign) but actually evaluates the old value. Notice that this applies to variables only, since generic expressions cannot be post-incremented.

(Var) In Java a variable alone may refer to different kinds of language entities: if it exists on the local variable environment, then it refers to a variable identifier and (Var-Local) applies; otherwise it may refer to a class member, being a shortcut for a select on `this` - which can either be an attribute field or an inner non-static class - and (Var-Member) applies.

(Sel) This rule deals with class member selection - either attribute fields or inner classes. For the sake of simplicity, the system looks up only final attribute fields because, given the complex control-flow of an Android application[20], reconstructing the life cycle of components and objects would require sophisticate data-flow analysis techniques (if decidable at all). Since most usages of attribute fields for defining constant data involve the final modifier, we argue our approach captures a wide enough range of applications. In case it refers to an inner class, the rules works anyway: left-hand expressions evaluates to an object of whatever address ω and type τ in which the identifier x is looked up regularly.

(New) This is responsible for creating new heap object references: the idea is to emulate the runtime heap memory by identifying objects by means of a unique natural number which resembles a memory address. Newly created objects are therefore fresh unique numbers attached to their respective type: variables are bound to their constant within the environment Γ, which makes multiple variables able to be bound to the same object, as it happens in by-reference languages like Java.

(Cond) Self-explanatory rules for the functional-like `if` construct.

(Call) Target method is looked up using the overload resolver as of definition 4.3.6, which needs the types of arguments to be reconstructed the method body is eventually evaluated in the environment Γ enriched with bindings for each method argument

(Subscript) This rule evaluates the n-th constant within the array literal, of course if the index expression can evaluate to an integer n. Evaluation of the array itself took place on array literal creation.

Evaluating Statements. Partial evaluation rules for statements are given in table 4.5. A detailed walk-through follows.

Table 4.5: Partial Evaluation Rules for Java Statements..

If-True

$$\frac{\widetilde{\tau}; \Gamma_0; \Omega_0; \Xi_0 \vdash e_1 \leadsto \ell[true] \rhd \Gamma_1; \Omega_1; \Xi_1 \qquad \widetilde{\tau}; \Gamma_1; \Omega_1; \Xi_1 \vdash st_1 \rhd \Gamma_2; \Omega_2; \Xi_2; \upsilon_1 \quad \overline{\Gamma} = \Gamma_1 \cup \{(x : \tau \leadsto \nu) \mid (x : \tau \leadsto \nu) \in \Gamma_2 \wedge x \in \Gamma_1\}}{\widetilde{\tau}; \Gamma_0; \Omega_0; \Xi_0 \vdash \texttt{if } e_1 \texttt{ then } st_1 \texttt{ else } st_2 \rhd \overline{\Gamma}; \Omega_2; \Xi_2; \upsilon_1}$$

If-False

$$\frac{\widetilde{\tau}; \Gamma_0; \Omega_0; \Xi_0 \vdash e_1 \leadsto \ell[false] \rhd \Gamma_1; \Omega_1; \Xi_1 \qquad \widetilde{\tau}; \Gamma_1; \Omega_1; \Xi_1 \vdash st_2 \rhd \Gamma_2; \Omega_2; \Xi_2; \upsilon_2 \quad \overline{\Gamma} = \Gamma_1 \cup \{(x : \tau \leadsto \nu) \mid (x : \tau \leadsto \nu) \in \Gamma_2 \wedge x \in \Gamma_1\}}{\widetilde{\tau}; \Gamma_0; \Omega_0; \Xi_0 \vdash \texttt{if } e_1 \texttt{ then } st_1 \texttt{ else } st_2 \rhd \overline{\Gamma}; \Omega_2; \Xi_2; \upsilon_2}$$

If-ϵ

$$\frac{\begin{array}{c}\widetilde{\tau}; \Gamma_0; \Omega_0; \Xi_0 \vdash e_1 \leadsto \epsilon \rhd \Gamma_1; \Omega_1; \Xi_1 \\ \widetilde{\tau}; \Gamma_1; \Omega_1; \Xi_1 \vdash st_1 \rhd \Gamma_2; \Omega_2; \Xi_2; \upsilon_1 \qquad \widetilde{\tau}; \Gamma_2; \Omega_2; \Xi_2 \vdash st_2 \rhd \Gamma_3; \Omega_3; \Xi_3; \upsilon_2 \\ \overline{\Gamma} = \Gamma_1 \cup \{(x : \tau \leadsto \epsilon) \mid (x : \tau) \in \Gamma_2 \wedge x \in \Gamma_1\} \cup \{(x : \tau \leadsto \epsilon) \mid (x : \tau) \in \Gamma_3 \wedge x \in \Gamma_1\}\end{array}}{\widetilde{\tau}; \Gamma_0; \Omega_0; \Xi_0 \vdash \texttt{if } e_1 \texttt{ then } st_1 \texttt{ else } st_2 \rhd \overline{\Gamma}; \Omega_3; \Xi_3; \epsilon}$$

Assign

$$\frac{(x : \tau) \in \Gamma \vee (x^F : \tau) \in \Gamma}{\widetilde{\tau}; \Gamma; \Omega; \Xi \vdash x \texttt{:=} e \rhd \Gamma, (x : \tau \leadsto \upsilon); \epsilon}$$

While-ϵ

$$\frac{\begin{array}{c}\widetilde{\tau}; \Gamma_0; \Omega_0; \Xi_0 \vdash e_1 \leadsto \upsilon_0 \rhd \Gamma_1; \Omega_1; \Xi_1 \\ \widetilde{\tau}; \Gamma_1; \Omega_1; \Xi_1 \vdash st \rhd \Gamma_2; \Omega_2; \Xi_2; \upsilon_1 \qquad \overline{\Gamma} = \Gamma_1 \cup \{(x : \tau \leadsto \epsilon) \mid (x : \tau) \in \Gamma_2 \wedge x \in \Gamma_1\}\end{array}}{\widetilde{\tau}; \Gamma_0; \Omega_0; \Xi_0 \vdash \texttt{while } e \texttt{ do } st \rhd \overline{\Gamma}; \Omega_2; \Xi_2; \epsilon}$$

Return

$$\frac{\widetilde{\tau}; \Gamma_0; \Omega_0; \Xi_0 \vdash e_1 \leadsto \upsilon \rhd \Gamma_1; \Omega_1; \Xi_1}{\widetilde{\tau}; \Gamma_0; \Omega_0; \Xi_0 \vdash \texttt{return } e \rhd \Gamma_1; \Omega_1; \Xi_1; \upsilon}$$

Decl

$$\frac{\widetilde{\tau}; \Gamma_0; \Omega_0; \Xi_0 \vdash e \leadsto \upsilon \rhd \Gamma_1; \Omega_1; \Xi_1}{\widetilde{\tau}; \Gamma_0; \Omega_0; \Xi_0 \vdash \tau\ x = e \rhd \Gamma_1, (x^F : \tau \leadsto \upsilon); \Omega_1; \Xi_1; \epsilon}$$

St-Expr

$$\frac{\widetilde{\tau}; \Gamma_0; \Omega_0; \Xi_0 \vdash e \leadsto \upsilon \rhd \Gamma_1; \Omega_1; \Xi_1}{\widetilde{\tau}; \Gamma; \Omega; \Xi \vdash e \rhd \Gamma_1; \Omega_1; \Xi_1; \epsilon}$$

Seq

$$\frac{\widetilde{\tau}; \Gamma_0; \Omega_0; \Xi_0 \vdash st_1 \rhd \Gamma_1; \Omega_1; \Xi_1; \upsilon_1 \qquad ; \Gamma_1; \Omega_1; \Xi_1 \vdash st_2 \rhd \Gamma_2; \Omega_2; \Xi_2; \upsilon_2}{\widetilde{\tau}; \Gamma_0; \Omega_0; \Xi_0 \vdash st_1 \texttt{;}\ st_2 \rhd \Gamma_2; \Omega_2; \Xi_2; \upsilon_2}$$

(Return) Statement rules do not evaluate to a constant: they rather *forward* a constant calculated by the `return` statement; all rules can effect the variable type environment Γ, though.

(Seq) The only rule propagating the effected environment while applying to different statements is (Seq), which is the defined as a typical *semicolon* combine rule.

(Assign) The main challenge here is dealing with assignments. The basic principle is that all variables could be potentially statically evaluable even when assignments are performed: the source of undecidability are loops and branches, not assignments on their own. Rule (Assign) therefore simply rebinds the left-hand variable in the Γ with the new constant and removing the final modifier, if present - this makes the system able to evaluate code blocks declaring non-final variables without initializer and assigning a value later, for example.

(If) When `if` branches are encountered, though, only those with a condition that can be evaluated successfully will not modify the way non-final variables are handled: depending on the boolean constant, either the variables within the `then` or `else` statement that are also defined in the outer scope will get updated within the output environment with the (possibly) new constant values.

(If-ϵ) In case the boolean condition cannot be evaluated, rule (If-ϵ) applies. It basically invalidates all variables defined in both inner and outer scopes that do not show the final modifier - i.e. those that appeared as left-values in assignments. This is the ratio behind rule (Assign) removing final modifiers: touched variables are subject to invalidation; while untouched ones are considered final, whether they have been explicitly declared as such or not[7].

(While) All loops are encodable by means of the `while` construct, hence the absence of `for` and `do..while` constructs in our model). Rule (While-ϵ) always invalidates all variables sharing both the outer and the inner scopes, similarly to what rule (If-ϵ) does. The reason for this conservative behaviour is that evaluating loops would require an interpreter model for Java featuring a set of sophisticate semantic rules - and all this only for dealing with trivial loops based on statically evaluable conditions: a possibility that is extremely unlikely.

(Decl) Declarations consist simply in binding the variable to its type and evaluated initializer expression within the environment Γ. Notably, the final

[7]This mechanism based on the propagation of the final modifier as a way for checking constantness resembles the const-checking system performed by the C++ compiler[5].

modifier is always used at declaration-time - all local variables are actually final as long as they are not assigned.

(St-Expr) This rule just applies the rule for expressions to a statement expression. Notice that this does not make the expression compute a value in an imperative language like Java - this just makes a non-void return value be discarded.

4.3.4 Type Reconstruction Rules

Rules for type reconstruction are tied to those for partial evaluation: the premise of both kinds of rule involves the same items but type reconstruction rules are the lonely responsible for new bindings to the Intent as well as component environments. Basically type reconstruction rules are the *main* rules - the entry point of the system: the whole Java program is in fact analyzed by those, which at times perform on-spot partial evaluation of expressions.

Type reconstruction rules for statements are given in table 4.6. We omit trivial rules that simply descend into inner sub-expressions and sub-statements and show only rule which are meaningful with respect to typing Intents and components: for example, the rule for `if-then-else` would just apply rules to the conditional expression, then to the two statement blocks forwarding environments as continuations as usual - that would have nothing to do with typing Intents or components, type-wise. Basically the system detects a number of Android API - i.e. method calls - and behaves accordingly: language constructs are not interesting here. While for partial evaluation rules most part of the inner complexity was due to interpreting Java constructs, operators, control-flow, ecc., when it comes to type reconstruction the complexity is detecting Android API calls and reconstructing information from local bits of information. Our system is designed for letting rules progressively add local information to a context that, by the end of the analysis, will contain all Intent and component types. Post-analysis reasoning on the resulting context will detect possible errors in inter-component communication, as shown below.

Type-checking Java. Another challenge of this kind of rules is that, being all method calls at the end of the day, a detection based on the method name would be naive, as it would generate possible false positives in case custom user code defines methods with the same name. The type of the object invoking the method must be inferred and the rule must apply only when such type is a subtype of the basic Android component supertypes (`Activity` and `Service`). This implies that a type checker for Java expressions is needed - and this in turn means that types of all variables, attributes, classes, method arguments must be checked: a full-featured Java type-checker, in other words.

As said above, we will not give rules for Java expressions - they're well known. But of course our implementation provides a full Java type checker by typing all

language construct on bottom of the type reconstruction rules that are peculiar to the system.

Non-statically Evaluable Intent Keys. Keep in mind that in this model, when the partial evaluation sub-system is not able to compute a constant for a string key of an Intent, it simply does not add that key to the Intent type. This case is not handled by the formal system for the sake of simplicity, but the `Lintent` implementation notifies that to the user and creates a special string key computed by hashing the non-evaluable expression and its program location. The ratio behind this is that in case non-statically evaluable expressions are used for computing string keys at runtime, then we argue there is a probability the that the same expression will be used in other parts of the program for computing the very same string key. Of course this does not belong to the world of exactness: that's why we did not formalize that with a typing rule - but it is worth mention nonetheless, since in real-world applications such an event may occur and has to be handled in some wise way even if it is beyond the the theoretical limits of computability.

Typing Statements. Notice that some Android API calls are detected by rules for statements and others for expressions. This apparent lack of symmetry is actually due to some methods having a return type and others being marked as `void`: `Intent.putExtra()` for example does not return a value, therefore calls are syntactically statement expressions - i.e. a production of the non-terminal statement; `Intent.getExtra()`, instead, returns something and is therefore an expressions.

Basically one group of rules are capable of reconstructing the type of Intents and a second group deals with API calls related to inter-component communication, thus they're related to the typing of components. Intent content types can be inferred by typing `putExtra()` overloaded methods and methods of the `get`T`Extra()` (where T is a supported type); component types can be reconstructed by putting together the information gathered when Activities and Services communicate.

We now propose a detailed description of the most meaningful rules for type reconstruction of expressions.

> (putExtra) A pair of rules deal with `putExtra()` calls over Intents. If the string key specified as first argument might already exists in the type reconstructed for the Intent, rule (putExtra-$\nexists$) applies: it adds no new key to the Intent type and it simply verifies that the type of the former-existing key is equal or a supertype of the newly-added value. Otherwise (putExtra-$\exists$) applies adding the new binding to the Intent type.
>
> (Decl) This rules is straightforward: it simply introduces new bindings to the variable type environment.
>
> (startActivityForResult) This is one of the core API calls of the whole Android system. The Intent argument and the request code are evaluated as

well as the `Activity` object invoking the method; this is typically srcthis but it could actually be any subclass of `Activity`. The Intent reference is the added to the outgoing request set of the component performing the method call.

(startActivity) Two analogous rules exist for this method call: depending on the type of object invoking the method, either (Activity.startActivity) or (Service.startActivity) applies. They're basically the same, except for the syntactic structure of the type of the activity and service components. Indeed, both rules effect the outgoing request part of the component type[8].

(setResult) This API call is used by Activities for replying to sender components a given result, i.e. an Intent. Keep in mind that the second argument is the result code - not to be confused with the request code. That's why it is ignored by the rule and it does not appear in any part of the Activity component type: only the Intent reference is recorded in the outgoing result part.

Typing Expressions. Expressions are basically API calls again, with a difference: they're not `void` method calls. Typing the `getIntent()` method call is one of the main challenges here. It can be treated as a static constructor of Intent, as it literally introduces a novel Intent object into the scope, from a strictly Java perspective. When encountered inside an `onCreate()` callback within an `Activity` component, it represents the incoming request; when inside an `onActivityResult()` within an `Activity`, it is bound to a request code forwarded by the framework to the callback as integer argument and it effects the incoming result part; else, when encountered while typing a `Service` component, it effects the incoming request of the Service component type.

Type reconstruction rules for expressions are shown in table 4.7. For the sake of simplicity we do not give rules for descending class methods as they would be trivial. Only one peculiar aspect of such rules is actually worth mentioning: the information on what method of a class is being typed must be carried along because it is needed for distinguishing different uses of the `getIntent()` method call. We explain them:

(Activity.getIntent-in-onCreate) When typing the `onCreate` method overridden by an Activity component, this rule applies and affects the incoming request part of the component type representing `this`. Notably, a new fresh reference to an undefined empty Intent type is introduced, since the method call actually behaves as a constructor that creates an Intent object with the default empty constructor.

[8]Due to page width, the effected component environment Ξ had to be redefined in the hypothesis rather than directly on the right hand of the thesis.

Table 4.6: Type Reconstruction Rules for Java Statements..

putExtra-$\nexists$

$$\frac{\begin{array}{c}\widetilde{\tau};\Gamma_0;\Omega_0;\Xi_0 \vdash e_0 \rightsquigarrow r \mapsto \circ : \texttt{Intent} \rhd \Gamma_1;\Omega_1;\Xi_1 \\ \widetilde{\tau};\Gamma_1;\Omega_1;\Xi_1 \vdash e_1 \rightsquigarrow \ell[\overline{s}] \rhd \Gamma_2;\Omega_2;\Xi_2 \qquad \widetilde{\tau};\Gamma_2;\Omega_2;\Xi_2 \vdash e_2 : \overline{\tau} \rhd \Gamma_3;\Omega_3;\Xi_3 \\ \Omega(r) = (\varrho)\{s_1:\tau_1,..,s_n:\tau_n\} \qquad \nexists i \in [1,n] \mid \overline{s} = s_i\end{array}}{\widetilde{\tau};\Gamma_0;\Omega_0;\Xi_0 \vdash e_0.\texttt{putExtra}(e_1,e_2) \rhd \Gamma_3;\Omega_3,(r \mapsto (\varrho)\{s_1:\tau_1,..,s_n:\tau_n,\overline{s}:\overline{\tau}\});\Xi_3}$$

putExtra-$\exists$

$$\frac{\begin{array}{c}\widetilde{\tau};\Gamma_0;\Omega_0;\Xi_0 \vdash e_0 \rightsquigarrow r \mapsto \circ : \texttt{Intent} \rhd \Gamma_1;\Omega_1;\Xi_1 \\ \widetilde{\tau};\Gamma_1;\Omega_1;\Xi_1 \vdash e_1 \rightsquigarrow \ell[\overline{s}] \rhd \Gamma_2;\Omega_2;\Xi_2 \qquad \widetilde{\tau};\Gamma_2;\Omega_2;\Xi_2 \vdash e_2 \rightsquigarrow \overline{\tau} \rhd \Gamma_3;\Omega_3;\Xi_3 \\ \Omega(r) = (\varrho)\{s_1:\tau_1,..,s_n:\tau_n\} \qquad \exists i \in [1,n] \mid \overline{s} = s_i \wedge \overline{\tau} \sqsubseteq \tau_i\end{array}}{\widetilde{\tau};\Gamma_0;\Omega_0;\Xi_0 \vdash e_0.\texttt{putExtra}(e_1,e_2) \rhd \Gamma_3;\Omega_3,(r \mapsto (\varrho)\{s_1:\tau_1,..,s_n:\tau_n\});\Xi_3}$$

Decl

$$\frac{\widetilde{\tau};\Gamma_0;\Omega_0;\Xi_0 \vdash e : \sigma \rhd \Gamma_1;\Omega_1;\Xi_1 \qquad \sigma \sqsubseteq \tau \qquad \widetilde{\tau};\Gamma_1;\Omega_1;\Xi_1 \vdash e \rightsquigarrow \upsilon \rhd \Gamma_2;\Omega_2;\Xi_2}{\widetilde{\tau};\Gamma_0;\Omega_0;\Xi_0 \vdash \tau\ x^F = e \rhd \Gamma_2,(x^F:\tau \rightsquigarrow \upsilon);\Omega_2;\Xi_2}$$

startActivityForResult

$$\frac{\begin{array}{c}\widetilde{\tau};\Gamma_0;\Omega_0;\Xi_0 \vdash e_0 : \widetilde{\tau} \rhd \Gamma_1;\Omega_1;\Xi_1 \\ \widetilde{\tau} \sqsubseteq \texttt{Activity} \qquad \widetilde{\tau};\Gamma_1;\Omega_1;\Xi_1 \vdash e_1 \rightsquigarrow r \mapsto \circ : \texttt{Intent} \rhd \Gamma_2;\Omega_2;\Xi_2 \\ \widetilde{\tau};\Gamma_2;\Omega_2;\Xi_2 \vdash e_2 \rightsquigarrow \ell[n] \rhd \Gamma_3;\Omega_3;\Xi_3 \qquad \lceil\widetilde{\tau}\rceil = \texttt{T} \\ \Xi(\texttt{T}) = \mathcal{I}_1 \to \mathcal{I}_2 \Longrightarrow \mathcal{L}_1 \hookleftarrow \mathcal{L}_2 \qquad \Xi_4 = \Xi_3,(\texttt{T}:\mathcal{I}_1 \to \mathcal{I}_2 \Longrightarrow \mathcal{L}_1 \cup \{(r,n)\} \hookleftarrow \mathcal{L}_2)\end{array}}{\widetilde{\tau};\Gamma_0;\Omega_0;\Xi_0 \vdash e_0.\texttt{startActivityForResult}(e_1,e_2) \rhd \Gamma_3;\Omega_3;\Xi_4}$$

Activity.startActivity

$$\frac{\begin{array}{c}\widetilde{\tau};\Gamma_0;\Omega_0;\Xi_0 \vdash e_0 : \widetilde{\tau} \rhd \Gamma_1;\Omega_1;\Xi_1 \\ \widetilde{\tau} \sqsubseteq \texttt{Activity} \qquad \widetilde{\tau};\Gamma_1;\Omega_1;\Xi_1 \vdash e_1 \rightsquigarrow r \mapsto \circ : \texttt{Intent} \rhd \Gamma_2;\Omega_2;\Xi_2 \\ \lceil\widetilde{\tau}\rceil = \texttt{T} \qquad \Xi(\texttt{T}) = \mathcal{I}_1 \to \mathcal{I}_2 \Longrightarrow \mathcal{L}_1 \hookleftarrow \mathcal{L}_2\end{array}}{\widetilde{\tau};\Gamma_0;\Omega_0;\Xi_0 \vdash e_0.\texttt{startActivity}(e_1) \rhd \Gamma_2;\Omega_2;\Xi_2,(\texttt{T}:\mathcal{I}_1 \to \mathcal{I}_2 \Longrightarrow \mathcal{L}_1 \cup \{(r,\cdot)\} \hookleftarrow \mathcal{L}_2)}$$

Service.startActivity

$$\frac{\begin{array}{c}\widetilde{\tau};\Gamma_0;\Omega_0;\Xi_0 \vdash e_0 : \widetilde{\tau} \rhd \Gamma_1;\Omega_1;\Xi_1 \qquad \widetilde{\tau} \sqsubseteq \texttt{Service} \\ \widetilde{\tau};\Gamma_1;\Omega_1;\Xi_1 \vdash e_1 \rightsquigarrow r \mapsto \circ : \texttt{Intent} \rhd \Gamma_2;\Omega_2;\Xi_2 \qquad \lceil\widetilde{\tau}\rceil = \texttt{T} \qquad \Xi(\texttt{T}) = \mathcal{I} \Longrightarrow \mathcal{L}\end{array}}{\widetilde{\tau};\Gamma_0;\Omega_0;\Xi_0 \vdash e_0.\texttt{startActivity}(e_1) \rhd \Gamma_2;\Omega_2;\Xi_2,(\texttt{T}:\mathcal{I}_1 \Longrightarrow \mathcal{I}_2 \cup \{r\})}$$

setResult

$$\frac{\begin{array}{c}\widetilde{\tau};\Gamma_0;\Omega_0;\Xi_0 \vdash e_0 : \widetilde{\tau} \rhd \Gamma_1;\Omega_1;\Xi_1 \\ \widetilde{\tau} \sqsubseteq \texttt{Activity} \qquad \widetilde{\tau};\Gamma_1;\Omega_1;\Xi_1 \vdash e_1 \rightsquigarrow r \mapsto \circ : \texttt{Intent} \rhd \Gamma_2;\Omega_2;\Xi_2 \\ \lceil\widetilde{\tau}\rceil = \texttt{T} \qquad \Xi(\texttt{T}) = \mathcal{I}_1 \to \mathcal{I}_2 \Longrightarrow \mathcal{L}_1 \hookleftarrow \mathcal{L}_2\end{array}}{\widetilde{\tau};\Gamma_0;\Omega_0;\Xi_0 \vdash e_0.\texttt{setResult}(e_1,e_2) \rhd \Gamma_3;\Omega_3;\Xi_3,(\texttt{T}:\mathcal{I}_1 \to \mathcal{I}_2 \cup \{r\} \Longrightarrow \mathcal{L}_1 \hookleftarrow \mathcal{L}_2)}$$

(ACTIVITY.GETINTENT-IN-ONACTIVITYRESULT) When typing the `onActivityResult` method overridden by an Activity component, instead, the incoming result part of the component is effected.

(SERVICE.GETINTENT) When the type of `this` reveals a Service, the incoming request part of the component type is effected.

(GETEXTRA) A single rule for integers stands as representative for the whole family of rules defined for each primitive type. In case the string key already exists, rule (GETEXTRA-$T-\exists$) (where T is the primitive type) applies and no new binding is inserted into the Intent type; otherwise rule (GETEXTRA-$T-\nexists$) applies and a new string key is bound to type T into the Intent type. This and rule (PUTEXTRA) are symmetric: the former populates Intent types as the user code puts data within Intents to be sent by sender components, the latter as user code extracts data from Intents received by recipient components.

(NEW-INTENT) A group of rules deal with Intent constructor invocation. Again the problem is to distinguish between different behaviours by resolving overloaded constructors: the 3 constructors shown here[9] are actually all unary constructors: the type of the argument discriminates between different semantics. In case the argument evaluates to a class object literal, it means the Intent is explicit; in case the argument evaluates to a string, it's implicit; in case it is an Intent, the copy constructor is invoked[10]. If the empty constructor is invoked, a new undefined Intent is created. Whatever the case, though, Intents are stored in the Ω environment, which reproduces the runtime heap memory for Intents only.

A couple of additional rules for special literals are shown as well: while the typing of ordinary literals is trivial, the type rules for the `null` literal and for the `.class` operator are given.

Keep also in mind that the Android API unfortunately offers several ways for doing things, often involving side effects and impure or unchecked programming conventions. For example, for setting the recipient of an Intent an user can create an explicit Intent by creating an empty Intent and then setting the recipient component type by name:

```
Intent i = new Intent();
i.setComponentName(new ComponentName("com.domain.app", "MyComponent");
```

[9]Much more Intent constructors exist in the Android API: the implementation deal with all of them, but here we show a few representatives for the sake of simplicity.

[10]According to the Android API documentation this creates a new Intent with the contents of the input one but there's no connection between them afterwards - it is not a reference to another Intent but a deep-copy of it [19].

Table 4.7: Type Reconstruction Rules for Java Expressions..

LIT-NULL

$$\frac{}{\widetilde{\tau};\Gamma;\Omega;\Xi \vdash \texttt{null} : \bot \rhd \Gamma;\Omega;\Xi}$$

LIT-CLASS

$$\frac{}{\widetilde{\tau};\Gamma;\Omega;\Xi \vdash \texttt{T.class} : \texttt{Class<T>} \rhd \Gamma;\Omega;\Xi}$$

ACTIVITY.GETINTENT-IN-ONCREATE

$$\frac{\widetilde{\tau};\Gamma_0;\Omega_0;\Xi_0 \vdash e_0 : \widetilde{\tau} \rhd \Gamma_1;\Omega_1;\Xi_1 \qquad \widetilde{\tau} \sqsubseteq \texttt{Activity} \\ \lceil\widetilde{\tau}\rceil = \texttt{T} \qquad \Xi(\texttt{T}) = \mathcal{I}_1 \to \mathcal{I}_2 \Longrightarrow \mathcal{L}_1 \hookleftarrow \mathcal{L}_2 \qquad \imath = (\texttt{?})\{\} \qquad r \in \mathbb{N}^* \texttt{ fresh}}{\widetilde{\tau};\Gamma_0;\Omega_0;\Xi_0 \vdash e_0.\texttt{getIntent}() : \texttt{Intent} \rhd \Gamma_1;\Omega_1,(r \mapsto \imath);\Xi_1,(\texttt{T} : \mathcal{I}_1 \cup \{r\} \to \mathcal{I}_2 \Longrightarrow \mathcal{L}_1 \hookleftarrow \mathcal{L}_2)}$$

ACTIVITY.GETINTENT-IN-ONACTIVITYRESULT

$$\frac{\widetilde{\tau};\Gamma_0;\Omega_0;\Xi_0 \vdash e_0 : \widetilde{\tau} \rhd \Gamma_1;\Omega_1;\Xi_1 \qquad \widetilde{\tau} \sqsubseteq \texttt{Activity} \\ \lceil\widetilde{\tau}\rceil = \texttt{T} \qquad \Xi(\texttt{T}) = \mathcal{I}_1 \to \mathcal{I}_2 \Longrightarrow \mathcal{L}_1 \hookleftarrow \mathcal{L}_2 \qquad \imath = (\texttt{?})\{\} \qquad r \in \mathbb{N}^* \texttt{ fresh}}{\widetilde{\tau};\Gamma_0;\Omega_0;\Xi_0 \vdash e_0.\texttt{getIntent}() : \texttt{Intent} \rhd \Gamma_1;\Omega_1,(r \mapsto \imath);\Xi_1,(\texttt{T} : \mathcal{I}_1 \to \mathcal{I}_2 \Longrightarrow \mathcal{L}_1 \hookleftarrow \mathcal{L}_2 \cup \{r\})}$$

SERVICE.GETINTENT

$$\frac{\widetilde{\tau};\Gamma_0;\Omega_0;\Xi_0 \vdash e_0 : \widetilde{\tau} \rhd \Gamma_1;\Omega_1;\Xi_1 \\ \widetilde{\tau} \sqsubseteq \texttt{Service} \qquad \lceil\widetilde{\tau}\rceil = \texttt{T} \qquad \Xi(\texttt{T}) = \mathcal{I}_1 \Longrightarrow \mathcal{I}_2 \qquad \imath = (\texttt{?})\{\} \qquad r \in \mathbb{N}^* \texttt{ fresh}}{\widetilde{\tau};\Gamma_0;\Omega_0;\Xi_0 \vdash e_0.\texttt{getIntent}() : \texttt{Intent} \rhd \Gamma_1;\Omega_1,(r \mapsto \imath);\Xi_1,(\texttt{T} : \mathcal{I}_1 \cup \{r\} \Longrightarrow \mathcal{I}_2)}$$

GETEXTRA-INT-$\exists$

$$\frac{\widetilde{\tau};\Gamma_0;\Omega_0;\Xi_0 \vdash e_0 \rightsquigarrow r \mapsto \circ : \texttt{Intent} \rhd \Gamma_1;\Omega_1;\Xi_1 \\ \widetilde{\tau};\Gamma_1;\Omega_1;\Xi_1 \vdash e_1 \rightsquigarrow \ell[\overline{s}] \rhd \Gamma_2;\Omega_2;\Xi_2 \qquad \widetilde{\tau};\Gamma_2;\Omega_2;\Xi_2 \vdash e_2 \rightsquigarrow \upsilon_2 \rhd \Gamma_3;\Omega_3;\Xi_3 \\ \Omega(r) = (\varrho)\{s_1 : \tau_1, .., s_n : \tau_n\} \qquad \exists i \in [1,n] \mid s_i = \overline{s} \wedge \tau_i = \texttt{int}}{\widetilde{\tau};\Gamma_0;\Omega_0;\Xi_0 \vdash e_0.\texttt{getIntExtra}(e_1, e_2) : \texttt{int} \rhd \Gamma_3;\Omega_3,(r \mapsto (\varrho)\{s_1 : \tau_1, .., s_n : \tau_n\});\Xi_3}$$

GETEXTRA-INT-$\nexists$

$$\frac{\widetilde{\tau};\Gamma_0;\Omega_0;\Xi_0 \vdash e_0 \rightsquigarrow r \mapsto \circ : \texttt{Intent} \rhd \Gamma_1;\Omega_1;\Xi_1 \\ \widetilde{\tau};\Gamma_1;\Omega_1;\Xi_1 \vdash e_1 \rightsquigarrow \ell[\overline{s}] \rhd \Gamma_2;\Omega_2;\Xi_2 \qquad \widetilde{\tau};\Gamma_2;\Omega_2;\Xi_2 \vdash e_2 \rightsquigarrow \upsilon_2 \rhd \Gamma_3;\Omega_3;\Xi_3 \\ \Omega(r) = (\varrho)\{s_1 : \tau_1, .., s_n : \tau_n\} \qquad \nexists i \in [1,n] \mid s_i = \overline{s}}{\widetilde{\tau};\Gamma_0;\Omega_0;\Xi_0 \vdash e_0.\texttt{getIntExtra}(e_1, e_2) : \texttt{int} \rhd \Gamma_3;\Omega_3,(r \mapsto (\varrho)\{s_1 : \tau_1, .., s_n : \tau_n, \overline{s} : \texttt{int}\});\Xi_3}$$

NEW-INTENT-UNDEF

$$\frac{r \in \mathbb{N}^* \texttt{ fresh}}{\widetilde{\tau};\Gamma;\Omega;\Xi \vdash \texttt{new Intent}() : \texttt{Intent} \rhd \Gamma;\Omega,(r \mapsto (\texttt{?})\{\});\Xi}$$

NEW-INTENT-EXPLICIT

$$\frac{\widetilde{\tau};\Gamma_0;\Omega_0;\Xi_0 \vdash e \rightsquigarrow \ell[\texttt{T.class}] \rhd \Gamma_1;\Omega_1;\Xi_1 \qquad r \in \mathbb{N}^* \texttt{ fresh}}{\widetilde{\tau};\Gamma_0;\Omega_0;\Xi_0 \vdash \texttt{new Intent}(e) : \texttt{Intent} \rhd \Gamma_1;\Omega_1,(r \mapsto (\texttt{T.class})\{\});\Xi_1}$$

NEW-INTENT-IMPLICIT

$$\frac{\widetilde{\tau};\Gamma_0;\Omega_0;\Xi_0 \vdash e \rightsquigarrow \ell[s] \rhd \Gamma_1;\Omega_1;\Xi_1 \qquad r \in \mathbb{N}^* \texttt{ fresh}}{\widetilde{\tau};\Gamma_0;\Omega_0;\Xi_0 \vdash \texttt{new Intent}(e) : \texttt{Intent} \rhd \Gamma_1;\Omega_1,(r \mapsto (\texttt{@}s)\{\});\Xi_1}$$

NEW-INTENT-COPYCONS

$$\frac{\widetilde{\tau};\Gamma_0;\Omega_0;\Xi_0 \vdash e \rightsquigarrow r \mapsto \circ : \texttt{Intent} \rhd \Gamma_1;\Omega_1;\Xi_1 \qquad \Omega(r) = \imath \qquad r' \in \mathbb{N}^* \texttt{ fresh}}{\widetilde{\tau};\Gamma_0;\Omega_0;\Xi_0 \vdash \texttt{new Intent}(e) : \texttt{Intent} \rhd \Gamma_1;\Omega_1,(r' \mapsto \imath);\Xi_1}$$

Or by setting the recipient component type by passing an Activity as context and a class object:

```
Intent i = new Intent();
i.setComponentName(new ComponentName(this, MyComponent.class);
```

Or, again, by setting the class name by means of a facility method call:

```
Intent i = new Intent();
i.setClass(this, MyComponent.class);
```

Or, finally, by invoking a third utility:

```
Intent i = new Intent();
i.setClassName("com.domain.app", "MyComponent");
```

It is clear that the approach of the Android API is a source of troubles for a system that is based on static analysis of the source code: dozens of rules should be given for every possible combination of calls and patterns - and that is clearly unfeasible in a formal specification. An implementation must although be able to deal with all these tricky situations

4.4 Post-Typing Checks

Before discussing such further checks we define a subtype relation between Intents:

Definition 4.4.1 (Intent Subtyping)**.** We define a subtype relation between Intent types such that, given two Intents $\imath_1 = (\varrho)\{s_1^1 : \tau_1^1..s_n^1 : \tau_n^1\}$ and $\imath_2 = (\varrho)\{s_1^2 : \tau_1^2..s_n^2 : \tau_m^2\}$, $\imath_1 \sqsubseteq \imath_2$ iff $\forall i \in [1, m].s_i^2 : \tau_i^2 \in \imath_1$.

After the typing phase, the information gathered by type rules and collected within environments can be used for performing some additional checks.

- *Consistency of requests.* Outgoing requests should be compatible with incoming requests. The former set is populated by analyzing calls to `startActivityForResult()` and `startActivity()` and the latter by inspecting `getIntent()` calls within `onCreate()` code: such sets are compared in such a way that all requests are handled by the recipient component and that all Intent contents retrieved by the receiver have actually been put by the sender. Formally, for all explicit Intents $\imath$ mentioned in the outgoing request set $\mathcal{L}_1$ of an Activity in the component environment Ξ, the incoming request set of the recipient component T mentioned in $\imath$ must contain at least one Intent $\imath'$ such that $\imath \sqsubseteq \imath'$. Anyway even the case $\imath \neq \imath'$ might point out a potential error: the recipient component might be forgetting to extract a given key from the Intent - it can't be statically decided, of course, because it belongs to the domain of the user intentions. Nonetheless, a warning should be raised for informing the user of this.

- *Completeness of request code handling.*. Incoming results must handle all request codes used as outgoing requests. The former by inspecting request codes associated to incoming Intents retrieved within `onActivityResult()` code and the latter set gets populated by analyzing calls to `startActivityForResult()`: such sets are compared in such a way that all request codes are handled correctly by the sender. Formally, for all Intents and non-null request code pair $(\imath, n)$ within the outgoing request set $\mathcal{L}_1$ of an Activity, there must exist a pair $(\imath', n)$ within the incoming results set $\mathcal{L}_2$ of the same Activity.

- *Consistency of results.*. Outgoing results of receiver components are subject to a comparison against incoming results of sender components. The former set gets populated by analyzing calls to `setResult()` and the latter by inspecting request codes associated to incoming Intents retrieved within `onActivityResult()` code: such sets are compared in such a way that, if the application code provides both sender and receiver components that are communicating, Intents replied back from the receiver are correctly handled by the sender, i.e. the contents retrieved by the sender have all been put the receiver who populated the Intent for replying. Formally, for all Intents $\imath$ within the outgoing result set $\mathcal{I}_2$ of an Activity T in the component environment Ξ, if there exists a sender component T' whose outgoing request set contained Intents addressed to T, then in the incoming result set $\mathcal{L}_2$ of the sender T' must exist an Intent $\imath'$ such that $\imath \sqsubseteq \imath'$. As for the consistency of request validation, non-strict equality might mean error-prone scenarios here as well and is notified by means of a warning to the user.

4.5 Implementation: `Lintent`

Our implementation of the Android Type Analyzer is a tool named `Lintent`. As anticipated in the preface, it is currently under development and available for download.

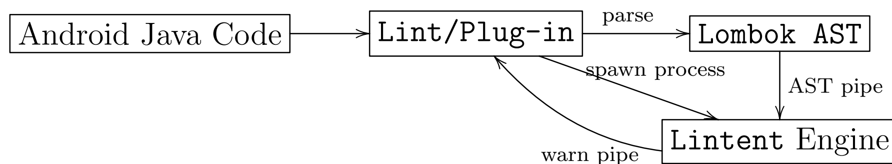

Figure 4.1: `Lintent` architecture

`Lintent` architecture is described in Figure 4.1 below. As anticipated, the tool is a `Lint` plug-in acting as a front-end for an engine program running as a separate process. The plug-in is written in Java and takes advantage of the built-in Java parser offered by `Lint`, which produces an Abstract Syntax Tree (AST) based on

Lombok AST [49]. Once parsing ends successfully, the engine process is spawned and starts receiving data from a pipe formerly created by the plug-in itself for interprocess communication. Our plug-in AST visitor simply serializes the program tree through the pipe and then waits for feedback from the engine process, hanging on a second pipe aimed at receiving warnings and messages to be eventually shown as issues by the Lint UI. The engine program is written in F# and does the real job: after deserializing the input program tree acquired from the AST pipe, it creates its own custom representation of the AST and performs the analysis.

The analysis consists in reconstructing the types of intents and components by means of a hybrid type-inference/partial-evaluation algorithm. Throughout the analysis, the engine communicates back with the Lint plug-in through the warn pipe, feeding back any issue worth to be prompted to the user via its built-in mechanisms of integration with Eclipse.

4.5.1 Engine Overview

The core of Lintent is the engine program written in F#. It consists of a number of modules:

Lexer and Parser The Java frontend of the plug-in uses Lombok AST API for rendering the parsed input program onto the AST pipe in a custom text format. Such data has to be parsed from the engine side via a Yacc-generated parser and a Lex-generated lexer for being translated into our custom Java AST representation. It is worth saying that parsing directly Java code could have been a possibility, removing the need to communicate between the Java and the F# parts; but that would have been more challenging due to a number of syntactic sugars and ambiguous grammar productions that Java, as any real-world language, shows: the Lint built-in parser does the job well, and just the AST provided by Lombok is translated into our own.

Custom Java AST Java AST is defined as a series of type definitions mixing classes and variants - and does not reflect the way Lombok AST is defined because such representation would be suboptimal for a functional language. Using variants for language expressions and statements is very common in compiler implementations written in ML-like languages [62], as opposed to object-oriented languages that typically rely on the *visitor* design pattern [65] for modeling ASTs. F# supports object-oriented programming as well, so a mixed approach has been used, gaining the best of the two worlds. Classes are useful for modeling macroscopic language constructs such as compilation units, classes, interfaces, methods and attributes: subtype polymorphism allows a lot of code reuse on similar data types. Variants are best suited for representing language statements and expressions, instead, thanks to the ability of performing pattern matching on them for implementing case-based

rules which can be placed in the same location of the source code and do not need to be spread around multiple source files, improving scalability and maintainability.

Auxiliary parsers A few additional Yacc-generated parsers are defined for a number of side operations. A parser for the output of `javap` exists as well as a tiny parser for Intent types annotated by the user by means of Java annotations.

Java type-checker A full-featured type-checker for Java expressions is implemented. As anticipated in section 4.3 this is highly needed by most Intent and component reconstruction algorithms for some important reasons:

1. when Android API calls are detected, recognizing method names alone would be unsafe: reconstructing the type of objects invoking the method and its arguments are needed for resolving possibly overloaded methods or constructors defined by the API and for ensuring no homonymous user-defined or third-party methods is being invoked;

2. in order to reconstruct the types of object injected into Intents, the type of arguments passed to `putExtra()` must be reconstructed for resolving which overload of `putExtra` is being invoked;

3. when the partial evaluation subsystem performs static computation of method calls, method bodies are needed: this means that overloading resolution is again needed for picking up the right method;

4. user code might use any kind of external `jar` or third party library, therefore object injected into Intents as well as any other data that is subject to partial evaluation could use those external packages: code must be fully type checked in the same way as the compiler does; the additional analysis our system performs is built on top of that.

This means that the types of every program variable and expression has to be reconstructed by simulating the Java type checker precisely as the Java compiler does: all classes, interfaces and methods must be typed because they could be used with Intents and components.

Intent and component type reconstruction. The core of the system consists in descending the AST and reconstruct a lot of type information by detecting relevant Android API calls as well as all Java constructs. This *first pass* populates a number of environments defined within the state of a monad (see section 4.6.2 below for details) and performs two main tasks: reconstructing the type of Intents and components, and partially evaluating code portions on-demand according to the needs of the type analysis. Technically speaking, it is not properly a type inference in the common sense because it does not involve unification and other mechanisms

that historically defined the notion of type inference itself [25]. It is rather a custom algorithm based on adding type information incrementally, exactly reproducing the flow of a strictly imperative API as Android's.

Security verification. `Lintent` performs an additional task as a *second pass* taking place after the type analysis described and formalized in this chapter. The scope of it is beyond the topic of this thesis and its mechanisms are therefore not shown in detail. Being a major component of our implementation, though, we believe it is at least worth mentioning and the reader may find a full insight of it in [11].

Android offers a permission system the Java compiler is completely oblivious of, since all permission information is encoded in terms of string literals used within the Java code and declared in the manifest. At the time of writing, even Android `Lint` does not perform any static check on permissions usage, thus leaving developers exposed, for instance, to run-time failures once the Android operating system detects a permission violation on some component interaction. `Lintent` performs a number of static checks over permissions usage, analyzing the application source code and the manifest permission declarations, and eventually warning the developer in case of potential attack surfaces for privilege escalation scenarios. As a byproduct of its analysis, `Lintent` is able to detect over-privileged or under-privileged applications, and suggest fixes.

4.5.2 Limitations and extensions

As of now our tool supports only activities and started services, while support for bound services is still under development and in a very preliminary stage. We plan to identify calls to API methods as `checkCallingPermissions()` to make our static analysis more precise. We are also investigating the possibility of developing a frontend to a decompiler as `smali` [2] or `ded` [29] to support the analysis of third-party applications.

4.6 Implementation Highlights

Our implementation is based on some advanced functional programming techniques which make its code scalable, succinct and highly reusable. All forms of polymorphism and nearly all F# language features have been used here or there throughout the code: object classes as well as pure records and variant types, pattern matching as well as dynamic dispatching on object-oriented hierarchies, sequence expressions and monads, and often a combination of all of these - everything except side-effects and imperative programming, unless strictly needed.

Most patterns, styles, conventions and the overall design used for coding `Lintent` are shared with the COBOL analyzer implementation, as many principles and so-

lutions are common to both. Peculiar highlights shown below do actually apply to both programs, indeed.

4.6.1 Support for Java Annotations

In real-world programs, some Intent types might not be reconstructible due to the user using reflection for computing labels or type themselves, for example. Similar and other weird cases may prevent `Lintent` from aiding the programmer with its set of checks just because an Intent type could not be inferred. To let the programmer specify an Intent type manually, `Lintent` supports special Java annotations:

```
@Intent("LABEL1 : int; LABEL2 : String")
Intent i = new Intent(this, Recipent.class);
i.putExtra(weirdExpressionComputingLABEL1(), 3);
i.putExtra("LABEL2", "foo");
```

Annotations must be put on an Intent initialization and the type specified within the annotation data refers to the heap reference created for that variable (refer to type rules (New-Intent) for details). Our system does not rely on variables but on heap references: therefore an Intent is not bound to a variable - but in Java it is, linguistically. Also Java is based on object references at runtime, and that's precisely what our system emulates statically, but for offering the programmer with a linguistic tool for typing Intents explicitly, annotations could not but be bound to object initializations - i.e variable bindings.

Additionally, annotations can be put on method parameters too and even variable declarations with no initializer - in general, on every language constructs that involves the introduction of a variable. Finally, type of Intent keys can be written using either the functional-like notation $x : \tau$ or the C-like syntax for variable declarations $\tau\ x$:

```
@Intent("double LABEL1, java.collections.generic.IList<String> LABEL2")
Intent i = new Intent(this, Recipent.class);
i.putExtra(weirdExpressionComputingLABEL1(), 3.5);
i.putExtra("LABEL2", new MySerializableList<String>());
```

The basic principles do not change, though: the Intent type newly created by the system gets the type specified within the annotation and it is eventually *locked*. This prevents further addition of any inferred keys and types to the Intent, including those that could be inferred wrongly - which is the reason why the user chose to annotate the type in the first place. The user must provide the full type, therefore, carefully paying attention to all the data that the code will put or extract into the Intent.

Keep in mind that Android does not support injection of any type into the Intent: only primitive types and `Serializable` are handled by the overloads of `putExtra`.

Our system though tracks the actual type of objects injected by using `putExtra`, thus the user can specify actual types or supertypes on the annotation - subtyping is naturally supported.

4.6.2 Monads and CPS

Rules in section 4.3 make extensive use of continuation-passing-style for affecting and forwarding environments in a *pure* paradigm with no need to introduce imperative-like side effects. This is typical of functional programming, of course, thus writing them in form of inference rules is as straightforward as writing them in form of functional code. As said, `Lintent` engine is written in F#, which supports monadic programming by means of a generalized re-writing subsystem called *computation expressions*[70]: by defining a set of primitives, the user can give custom semantics to the basic constructs of the computation expression sub-language. Identifier binding, sequential combine, `for` loops and other atomic language terms can be redefined, for example, for reproducing the behaviour of the state monad[69]:

```
type builder () =
    member m.Delay f : M<'s, 'a> = f ()
    member m.Return x : M<'s, 'a> = fun s -> (x, s)
    member m.ReturnFrom f : M<'s, 'a> = fun s -> f s
    member m.Bind (e, f) : M<'s, 'b> = fun s -> let (r, s') = e s in f r s'
    member m.Zero () : M<'s, unit> = fun s -> ((), s)
    member m.TryFinally (e, fin) : M<'s, 'a> = fun s -> try e s finally fin ()

    member m.TryWith (e : M<'s, 'a>, catch : exn -> M<'s, 'a>) : M<'s, 'a> =
        fun s -> try e s with exn -> catch exn s

    member m.For (seq, f) : M<'s, unit> =
        fun s -> ((), Seq.fold (fun s x -> let (r, s') = f x s in s') s seq)

    member m.Combine (e1, e2) : M<'s, 'a> =
        fun s -> let ((), s') = e1 s in e2 s'
```

Class `builder` contains a method definition for each re-definable computation expression construct. Types have been annotated for the sake of readability even if F# is capable of fully infer these functions. Every primitive is a polymorphic function over the state of the monad $'s$ in such a way that the same builder can be used for different state types.

Usage of the monad is simple and makes easy the implementation of typing rules over Java constructs which have recursive terms. Take the following snippet for a type-checker:

```
// Java type representation
type type = Ty_Int | Ty_Bool | ..
```

```
// state monad instance
let M = new builder ()

let rec typecheck_statement st =
  M {
    match st with
        | IfThenElse (e1, st1, st2) ->
            let! t1 = typecheck_expr e1
            if t1 <> Ty_Bool then raise Type_error
            let! () = typecheck_statement st1
            let! () = typecheck_statement st2
            return ()

        | // other statement union cases
  }

and typecheck_expr e =
  M {
    match e with
        | PlusBinOp (e1, e2) ->
            let! t1 = typecheck_expr e1
            let! t2 = typecheck_expr e2
            match t1, t2 with
                | Ty_Int, Ty_Int -> () // ok
                | _              -> raise Type_error
            return Ty_Int

        | // other expression union cases
  }
```

Despite resembling imperative programming in the name of constructs, the state monad in fact emulates a *stateful* paradigm while actually making the F# compiler generate purely functional code. Re-written functions are those whose body appears inside `M ...` : this tells the compiler to translate the computation expression within and the resulting code will lambda-abstract a further parameter - the state. In other words, if $e : \tau$ then `M` $\{\ e\ \} : \sigma \rightarrow \tau$ where σ is the state type. Due to the state monad definition, state is constantly lambda-abstract and applied by the implementation of primitives, hence the need to not pass as it argument it or to not bind it as result when monadic functions are called. In other words, this mechanism emulates a continuation-passing-style in a transparent way, relieving the programmer from the task of constantly forward the state thoroughly.

Analysis functions for each language non-terminal would have a lot of arguments and would output a big tuple if written in plain functional code. As far as input is concerned, looking at our tying rule premises, we distinguished between two kinds of

entities: pure and effected entities. Environments are effected, while for example the type of `this` $\widetilde{\tau}$ belongs to the pure set; another information that is pure is the name of the method we're currently analyzing - which does not appear in the formalization for simplicity reasons but is needed by the implementation for discriminating between `getIntent()` calls within method `onCreate` and `onActivityResult`.

Monad State. Effected environments have been put into the monad state:

```
type state =
  {
    var_decls  : env<id,      var_decl>
    classes    : env<fqid,    class_signature>
    intents    : env<address, intent>
    components : env<fqid,    componentt>
  }
```

The implementation has one more environment that the formalization: the class environment, which stores all classes and interfaces of the the program in their AST representation. This is needed because the lookup operators need to lookup somewhere - and the program carries along this class storage within the monad state.

Types used in the definition are:

- `env` is the environment type having two type parameters: the type of symbols and the type of values mapped by symbols;
- `id` is the type of identifiers;
- `fqid` is the type of fully-qualified identifiers;
- `var_decl` is the type of variable type judices $x : \tau \rightsquigarrow \upsilon$;
- `class_signature` is the super-type of classes and interfaces within the AST;
- `address` is the type of addresses of heap references $\omega : \tau$;
- `intent` is the type of Intents;
- `component` is the type of components ξ.

Pure Context. Other pure information are passed to analysis functions by means of a record called `context`:

```
type context =
  {
    this_ty            : J.ty
    current_method     : J.methodd option
  }
```

In the actual implementation a few of other data is stored within the monad state and the context record, which we omitted here for not delving into complex implementation architecture details.

4.6.3 Active Patterns as API Detectors

Detecting Android API calls, as already said, is not only a matter of matching the name of a method - that would be naive. The type of the object invoking the method must be inferred and possible overload resolution must occur for selecting the correct method among many with the same name.

Typically when writing a typing function (or in general a function dealing with language term processing) in an ML-like language, pattern matching among the many variants of each term type is used. Plain pattern matching is not powerful enough though for performing tasks that are more complex than matching terms or method names - as we said, at least the type of the calling object should take place to the match as well.

F# offers a powerful language feature called *Active Patterns*[70] which is elegantly integrated within the classic pattern matching ML and lets the user defined custom *virtual* patterns which can be used as regular patterns but perform custom code for actually performing the matching at runtime. In other words, the user code must deal with whatever comparison for deciding whether the Active Pattern matches or not given the expression being matched.

`Lintent` implementation make extensive use of Active Patterns for letting the analysis code perform apparently regular pattern matching over statement and expression terms, while actually special detection code take place - and such code performs not only the method name match but also the type checking of the object invoking it and of arguments.

A simple example:

```
module Detector =
    let (|Ty|_|) tyname envs ty =
        if TJ.qualify_ty envs ty = J.Ty_SQId tyname then Some ()
        else None

    module Intent =

        let (|Ty|_|) = (|Ty|_|) "android.content.Intent"

        let (|New|_|) envs = function
            | E (J.New (Ty envs, ([], args))) -> Some args
            | _                               -> None
```

Note that the active pattern `(|Ty|_|)` defined within submodule `Detector.Intent` is not recursive but is just shadowing the former definition in the outer module

`Detector`. The first active pattern `(|Ty|_|)` detects whether 'tyname' is a subtype or equal to 'ty' within environments 'envs', while the second is curried over the first and checks for the Intent type[11].

Code below can use the active pattern `Detector.Intent.New` for pattern matching calls to Intent constructors against Java expressions:

```
module D = Detector

let analyze_expr (e : J.expr) =
    match e with
        | D.Intent.New [e]      -> // perform checks on unary constructor invocation
        | D.Intent.New [e1; e2] -> // perform checks on binary constructor invocation
```

4.7 Experimental Results

With `Lintent` in alpha stage of development we have been able to run tests over a number of Android programs and snippets. As already said, our system is implemented as an ADT Lint plug-in and performs a number of validations that goes beyond the scope of Intent and component type reconstruction (permission checking against privilege escalation, above all). We'll stay in-topic here, though, and will strictly discuss about the type reconstruction feature, avoiding considerations on other aspects.

We ran several kinds of tests:

1. apps belonging to the official samples included in the Android SDK: some of these make wide use of implicit Intents, some others are more based on explicit ones - both present typical Android patterns, though, thus they're meaningful test-wise. For the sake of fair statistics, we picked samples from both groups, though it is clear that the benefits of our system are largely intended for applications involving a heavy use of explicit inter-component communication - how else could `Lintent` notify the user of certain ill-typed Intents or components if it cannot inspect them?

2. tests written by us meant for stressing the system in a way that probably no real-world application ever would. Think for example at who would put 4 nested components within a component or compute weird Intent labels by means of Intent results - similar scenarios are surely possible, but no code we had the opportunity to review has even come close to that.

[11]This snippet is taken from the actual implementation, therefore it uses several data types and utility functions that are defined by the many modules is made of. There is no need to delve into technical details: the overall *flavour* of Active Patterns usage is more important.

3. real-world apps from the open-source community or relevant portions of them: testing those of course might be more meaningful for testing the security-oriented features of `Lintent` rather that Intent typing. Reasonably, if an app gets released then it would likely work, and if it works then it means Intents and components are ok, otherwise it would have crashed already while being tested by the developer himself. We believe anyway that it is rather interesting to sample how many existing and running apps deal with Intents correctly and are bug-free.

One last consideration before showing results: during these test sessions we used `Lintent` as a code analyzer for validating *existing* apps - but we discovered that this is not how it performs at its best. Our tool behaves like an advanced type-checker, thus it is best used as an assistant for the developer, aiding him/her in finding and fixing issues in the code *while he/she writes it*. This was of course a rather qualitative consideration that cannot be brought by evidences and numbers, but can only be experienced while actually writing an app.

Effectiveness of Intents and components type Reconstruction.

- 85% of warning and error messages output by ADT Lint and originating from our `Lintent` plug-in are related to Intent and component mistyping
- apps define a number of Intents typically proportional to the number of components, which is up to 30% of the non-trivial code of the whole program
- apps heavily based on GUI typically define a lot of Activities and therefore a lot of explicit Intents: these are the applications benefitting most from `Lintent`. In such kind of apps, up to 75% of Intents are explicit and sent to components which are components of the app itself - i.e. the analyzer knows the code and can verify Intent usage on the recipient side.

  This implies a strong qualitative property over `Lintent`: it is mostly useful when both sender and recipient component code are available for analysis. Therefore, apps communicating mostly via implicit Intents addressed to system services cannot be checked effectively because *post-typing checks do not have enough information* for making meaningful comparisons between Intents that the sender and receiver components deal with.
- 95% of apps relies on external or third-party libraries (i.e. not belonging to the Java or the Android SDK), hence the analyzer needs to know their class, interface and method signatures in order to type-check code and successfully reconstruct Intents and components involved.This implies that users must provide `jar` files to `Lintent` in order to make it able to inspect imports in-depth. The type resolution algorithm never fails, though, and won't simply resolve

methods and other members within unknown classes in case no `jar` is provided; and, as far types are involved, the class name itself is kept as type if needed.

We observed that up to 35% of type occurrences within apps for which no external library `jar` had been provided to `Lintent` are successfully typed. This is due to the extensive use of methods defined in those external libraries, naturally. In other words, plain type names occur in the code, of course, but less than method calls.

- under 8% of apps use implicit Intents for communicating with own internal components; these include two cases:

  1. apps that define own Services or Service-like Activities[12] and some other component within the app itself is client of that Service;
  2. apps that misuse implicit-Intent-based mechanisms for inter-component communication: this is an issue related to programming *style* but it underlines bad habits in designing application code and very little knowledge of the Java language.

- apps which are basically clients of a system or third-party Service are based on mostly implicit Intent passing: for this kind of app, Intent checking is uninformative possibly up to 85% of all Intents involved (assuming, pessimistically, that a 15% minimum is involved within basic UI communication)

Effectiveness of Partial Evaluation.

- up to 95% of apps rely on constant keys for Intents, action strings and component recipient classes. Use of complex non-statically-evaluable algorithms for computing one of the bits above is quite unusual in the great majority of programs.

- what is less unusual is the dynamic structure of Intents: up to 30% of apps use a few Intent entries as control data, in order to make the recipient able to perform different tasks according to the presence of a special Intent data-less key or according to the value of a given integer or boolean key. Even a pretty simple app like the `Notepad` included within the Android SDK Tutorial makes use of such technique: the recipient component inspects a special Intent key and, depending on its existence within the received Intent, it performs different tasks.

  We might discuss whether this is or not a reasonable programming convention and how developers could avoid such error-prone approaches by using the Java

[12]Sometimes Activities behave like Services but have a visual component, hence they cannot inherit the `Service` class.

language and the Android API more carefully - but that's not the point of an analyzer. A lot of existing code out there behave like that: and `Lintent` is supposed to offer some degree of aid to developers used to such techniques, even if arguable. Admittedly, our system does not help much in this respect unfortunately: a warning is prompted in case the sender component does not put all the entries the recipient component will retrieve (refer to post-typing check in section 4.4, but understanding the dynamic behaviour of the recipient code by inspecting `if`-branches and Intent extras usage would require some form of data-flow analysis which is beyond the capabilities of our partial evaluation sub-system.

- about 40% of apps rely on non-final bits for Intent keys, action strings, etc. which are actually statically evaluable: this is a symptom of a bad programming style based on declaring uninitialized non-final variables or class attributes, eventually setting them to actually immutable values via assignment. Most cases are based on simple computations of key strings and our system is capable of evaluating them statically anyway, since no conditional branches or loops which would invalidate the evaluation are involved (refer to typing rules (If) and (While) in section 4.3.3).

We also observed that no inter-component communication issues affect apps belonging to the Android SDK Tutorial that have been tested - which likely means that Google developers clearly paid attention in providing error-free samples. Other third-party apps we have tested had little to no issues as well (below 3% of Intents involved overall).

This does not mean that `Lintent` is useless though: there might exist dozens of apps in the Google Play Store that could have not been as heavily debugged. Our system is similar to the type-checker of a programming language: users may develop even without it and invest a lot of time in debugging untyped code before releasing an application; having a compiler that notifies the user with type errors and warning, though, alleviates the developer from possibly hundreds of man-hours of debugging and testing.

Intent management and component inter-communication represent a sensible part of Android development and we believe that our type analysis makes the writing of those parts much safer and faster, possibly reducing debugging time of Intent-related code to zero.

Speaking of rough averages, a scenario that would best benefit of our system is:

- a stand-alone non-Service-based application code dealing with Intents for 20% of its non-trivial code, where
- roughly 80% of such Intents are explicit, then
- a developer would spend about 20% of its debugging time in fixing Intent passing, thus

- 16% of the debugging time would be saved by `Lintent`.

Even if we approximate such result down to 10% due to some Intents not being reconstructible or other limitations of the system, we believe that is still a valuable benefit for software houses as well as for independent developers releasing open-source apps.

4.8 Future Work

As stated in the introduction, the system proposed here is just the core typing module of a wider static analysis framework that offers security validation and permission checking too [11]. `Lintent` implementation already performs such checks, as the reader can see from the source code of the application.

Another major topic is already under research and development: integrating a type system for information-flow security to the existing features, somewhat resembling other DLM-based[13] systems such as JIF [58][23] and JLift [44] for Java, but with a more light-weight approach from the user perspective and fully oriented to Android. For keeping intact the user-experience `Lintent` brings, being an ADT Lint plug-in that relies on an external Java parser and operates together with possibly dozens of other analyzers, we had to host all DLM types and constructs on Java annotations - which is a pretty hard design task on its own. Another important feature is that all default behaviours of the type system fall back to transparent rules, in such a way that a totally unannotated application would compile as usual - other systems like JIF do not follow this principle and are therefore considered hard to learn and understand for the casual programmer. We believe that a system designed for Android should be easily and effectively used also by the many non-professional developers writing apps nowadays.

[13]The *Decentralized Label Model* is a well-known system based on static type analysis for controlling information flow in systems with mutual distrust and decentralized authority [57].

Bibliography

[1] The opencobol project. http://www.opencobol.org/.

[2] Smali: An assembler/disassembler for android's dex format. http://code.google.com/p/smali/.

[3] Wikipedia: Android operating system. http://en.wikipedia.org/wiki/Android_(operating_system).

[4] Programming language - cobol, ansi x3.23-1985 edition, 1985.

[5] Iso/iec 14882:1998: Programming languages - c++, September 1998.

[6] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[7] et al. Alan Roger. Cobol: Continuing to drive value in the 21st century. *Datamonitor*, November 2008.

[8] A. W. Appel. Cambridge University Press, 1992.

[9] A. Armando, G. Costa, and A. Merlo. Formal modeling and verification of the android security framework. In *TGC2012*, pages xx–xx, 2012. To Appear.

[10] M. G. J. Van Den Brand, J. Heering, P. Klint, and P. A. Olivier. Compiling language definitions: the asf+sdf compiler. *ACM Transactions on Programming Languages and Systems*, 24(4):334–368, 2002.

[11] Michele Bugliesi, Stefano Calzavara, and Alvise Spanò. Taming the android permissions system, by typing. Submitted.

[12] Carlos Camarão and Lucília Figueiredo. Type inference for overloading without restrictions, declarations or annotations. In *Fuji International Symposium on Functional and Logic Programming*, pages 37–52, 1999.

[13] Avik Chaudhuri. Language-based security on Android. In *PLAS*, pages 1–7, 2009.

[14] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in Android. In *MobiSys*, pages 239–252, 2011.

[15] Noam Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2:113–124, 1956.

[16] James R. Cordy. The txl source transformation language. *Sci. Comput. Program.*, 61(3):190–210, 2006.

[17] Google Corp. Adt lint. `http://tools.android.com/tips/lint`.

[18] Google Corp. Android platform sdk. `http://developer.android.com/reference/packages.html`.

[19] Google Corp. Android sdk - intent. `http://developer.android.com/reference/android/content/Intent.html#Intent(android.content.Intent)`.

[20] Google Corp. Managing the activity lifecycle. `http://developer.android.com/training/basics/activity-lifecycle/index.html`.

[21] Microsoft Corp. Windows presentation foundation. `http://msdn.microsoft.com/it-it/library/aa970268.aspx`.

[22] Oracle Corp. The java language specification - java se 7 edition. `http://docs.oracle.com/javase/specs/jls/se7/html/`.

[23] A. Myres D. Zhang, O. Arden. Jif: Java + information flow. `http://www.cs.cornell.edu/jif/`.

[24] Gartner Research Vice President Dale Vecchio. Impact of generational it skills shift on legacy applications. 2007.

[25] Luís Damas and Robin Milner. Principal type-schemes for functional programs. In *POPL*, pages 207–212, 1982.

[26] Dominic Duggan, Gordon V. Cormack, and John Ophel. Kinded type inference for parametric overloading. *Acta Inf.*, 33(1):21–68, 1996.

[27] William Enck. Defending users against smartphone apps: Techniques and future directions. In *ICISS*, pages 49–70, 2011.

[28] William Enck, Peter Gilbert, Byung gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*, pages 393–407, 2010.

[29] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. A study of Android application security. In *USENIX Security Symposium*, 2011.

[30] William Enck, Machigar Ongtang, and Patrick Drew McDaniel. Understanding android security. *IEEE Security & Privacy*, 7(1):50–57, 2009.

[31] C. Hankin F. Nielson, H.R. Nielson. *Principles of Static Analysis.* Springer Verlag, 1999.

[32] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *ACM Conference on Computer and Communications Security*, pages 627–638, 2011.

[33] Elli Fragkaki, Lujo Bauer, Limin Jia, and David Swasey. Modeling and enhancing Android's permission system. In *ESORICS*, volume 7459 of *Lecture Notes in Computer Science*, pages 1–18, 2012. To appear.

[34] Inc. Free Software Foundation. Bison reference manual. `http://www.gnu.org/software/bison/manual/bison.pdf`.

[35] Adam P. Fuchs, Avik Chaudhuri, and Jeffrey S. Foster. Scandroid: Automated security certification of android applications, 2009. Technical report, University of Maryland.

[36] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip Wadler. Type classes in haskell. In *ESOP*, pages 241–256, 1994.

[37] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism sdf-rreference manual. *SIGPLAN Not.*, 24(11):43–75, 1989.

[38] Richard C. Holt. WCRE 1998 most influential paper: Grokking software architecture. In *WCRE (Working Conference on Reverse Engeneering)*, pages 5–14. IEEE, 2008.

[39] IBM. Cobol z/OS language reference. Website, 2009. `http://publib.boulder.ibm.com/infocenter/pdthelp/v1r1/index.jsp?topic=/com.ibm.debugtool.doc_7.1/eqa7rm0293.htm`.

[40] Ernst jan Verhoeven. *COBOL island grammars in SDF.* PhD thesis, 2000.

[41] Stephen C. Johnson. Yacc: Yet another compiler-compiler. 32, 1975.

[42] Wayne Kelly and Australia Queensland University of Technology, Birsbane. The gardens point parser generator (gppg). `http://plas.fit.qut.edu.au/gppg/`.

[43] Brian W. Kernighan and Dennis Ritchie. *The C Programming Language, Second Edition.* Prentice-Hall, 1988.

[44] D. King. Jlift. `http://siis.cse.psu.edu/jlift/jlift.html`.

[45] Steven Klusener and Ralf Lämmel. Deriving tolerant grammars from a base-line grammar. In *ICSM*, page 179. IEEE Computer Society, 2003.

[46] Tobias Kuipers and Leon Moonen. Types and concept analysis for legacy systems. In *IWPC*, pages 221–230. IEEE Computer Society, 2000.

[47] R. Lämmel and C. Verhoef. Semi-automatic grammar recovery. *Software - Practice & Experience*, 31(15):1395–1438, December 2001.

[48] Michael E. Lesk and Eric Schmidt. Lex - a lexical analyzer generator. *Computing Science TR*, 39, October 1975.

[49] Project Lombok. Lombok ast. `http://projectlombok.org/api/lombok/javac/package-summary.html`.

[50] Amiya Kumar Maji, Fahad A. Arshad, Saurabh Bagchi, and Jan S. Rellermeyer. An empirical study of the robustness of inter-component communication in android. In *DSN*, pages 1–12, 2012.

[51] L. Moonen M.G.J. van den Brand, A.S. Klusener. Generalized parsing and term rewriting: Semantics driven disambiguation. *Electronic Notes in Theoretical Computer Science*, 82, Issue 3:575–591, 2003.

[52] Sun Microsystems. Javacc. `https://javacc.dev.java.net/`.

[53] Leon Moonen. Generating robust parsers using island grammars. In *Proceedings of the 8th Working Conference on Reverse Engineering*, pages 13–22. IEEE Computer Society Press, October 2001.

[54] Leon Moonen. Generating robust parsers using island grammars. In *WCRE (Working Conference on Reverse Engeneering)*, 2001.

[55] Leon Moonen. Lightweight impact analysis using island grammars. In *IWPC*, pages 219–228. IEEE Computer Society, 2002.

[56] Leon Moonen. Exploring software systems. In *ICSM*, pages 276–280. IEEE Computer Society, 2003.

[57] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *SOSP*, pages 129–142, 1997.

[58] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Trans. Softw. Eng. Methodol.*, 9(4):410–442, 2000.

[59] Institut national de recherche en informatique et automatique (INRIA). Ocaml language reference manual. `http://caml.inria.fr/distrib/ocaml-3.12/ocaml-3.12-refman.pdf`.

[60] Chris Parnin, Christian Bird, and Emerson R. Murphy-Hill. Java generics adoption: how new features are introduced, championed, or ignored. In *MSR*, pages 3–12, 2011.

[61] Terence J. Parr and Russell W. Quong. Antlr: A predicated-ll(k) parser generator. *Software—Practice and Experience*, 25(7):789–810, 1995.

[62] B. C. Pierce. The MIT Press, 2004.

[63] Bernard Pinon. A cobol parser. `http://mapage.noos.fr/~bpinon/a_cobol_parser.htm`.

[64] Erik Post. Island grammars in asf+sdf. Master's thesis, University of Amsterdam, 2007.

[65] C. Scott Ananian Scott Hudson, Frank Flannery. Javacup. `http://www.cs.princeton.edu/~appel/modern/java/CUP/`.

[66] Alvise Spanò, Michele Bugliesi, and Agostino Cortesi. Type-flow analysis for legacy cobol code. In *ICSOFT (2)*, pages 64–75, 2011.

[67] Alvise Spanò, Michele Bugliesi, and Agostino Cortesi. *Typing Legacy COBOL Code*. Springer-CCIS, 2012.

[68] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 2000.

[69] Nikhil Swamy, Nataliya Guts, Daan Leijen, and Michael Hicks. Lightweight monadic programming in ml. In *ICFP*, pages 15–27, 2011.

[70] D. Syme. The f# 3.0 language specification. `http://research.microsoft.com/en-us/um/cambridge/projects/fsharp/manual/spec.html`.

[71] Nikita Synytskyy, James R. Cordy, and Thomas R. Dean. Robust multilingual parsing using island grammars. In *CASCON*, pages 266–278. IBM, 2003.

[72] Adrian Thurston. *A Computer Language Transformation System Capable of Generalized Context-Dependent Parsing*. PhD thesis, Queen's University, 2008-12-15 21:01:24.236, 2008.

[73] Adrian D. Thurston and James R. Cordy. A backtracking lr algorithm for parsing ambiguous context-dependent languages. In Hakan Erdogmus, Eleni Stroulia, and Darlene A. Stewart, editors, *CASCON*, pages 39–53. IBM, 2006.

[74] Masaru Tomita. Lr parsers for natural languages. *COLING-84*, pages 354–357, 1984.

[75] Mark van den Brand, Alex Sellink, and Chris Verhoef. Obtaining a cobol grammar from legacy code for reengineering purposes. 1997.

[76] Arie van Deursen and Tobias Kuipers. Building documentation generators. In *ICSM*, pages 40–49, 1999.

[77] Arie van Deursen and Leon Moonen. Type inference for cobol systems. In *WCRE (Working Conference on Reverse Engeneering)*, pages 220–230, 1998.

[78] Arie van Deursen and Leon Moonen. Understanding cobol systems using inferred types. In *IWPC*. IEEE Computer Society, 1999.

[79] Arie van Deursen and Leon Moonen. Exploring legacy systems using types. In *WCRE (Working Conference on Reverse Engeneering)*, pages 32–41, 2000.

[80] Arie van Deursen and Leon Moonen. An empirical study into cobol type inferencing. *Sci. Comput. Program.*, 40(2-3):189–211, 2001.

[81] Arie van Deursen and Leon Moonen. Documenting software systems using types. *Sci. Comput. Program.*, 60(2):205–220, 2006.

[82] Eelco Visser. Scannerless generalized-lr parsing. Technical Report P9707, Programming Research Group, University of Amsterdam, 1997.