From Polymorphic Subtyping to CFL Reachability:
Context-Sensitive Flow Analysis Using Instantiation
Constraints

Microsoft Research Technical Report MSR-TR-99-84

Manuel Fähndrich    Jakob Rehof    Manuvir Das

March 2000
Abstract

We present a novel approach to computing context-sensitive flow of values through procedures and data structures. Our approach combines and extends techniques from two seemingly disparate areas: polymorphic subtyping and interprocedural dataflow analysis based on context-free language reachability. The resulting technique offers several advantages over previous approaches: it works directly on higher-order programs, provides demand-driven interprocedural queries, and improves the asymptotic complexity of a known algorithm based on polymorphic subtyping from $O(n^3)$ to $O(n^3)$ for computing all queries. For intra-procedural flow restricted to equivalence classes, our algorithm yields linear inter-procedural flow queries.
Contents

1 Introduction .................................................. 3
   1.1 Constraint Copying Methods ........................................... 5
   1.2 A New Method Based on Instantiation Constraints ......................... 5
   1.3 Situating the Analyses .................................................. 7
   1.4 Context-Sensitivity in Higher-Order Programs .............................. 8
   1.5 Overview .................................................................. 10
   1.6 Definitions ............................................................... 11

2 Flow Polymorphism ................................................. 15
   2.1 A Copy-based System ..................................................... 15
      2.1.1 Polymorphic Constrained Types ....................................... 15
      2.1.2 Type Rules ............................................................. 16
      2.1.3 Flow Graphs ............................................................ 18
      2.1.4 Flow Relation ........................................................... 19
      2.1.5 Example ................................................................. 19
   2.2 A CFL-based System ...................................................... 20
      2.2.1 Polymorphic Types ..................................................... 21
      2.2.2 Type Rules ............................................................. 21
      2.2.3 Flow Graphs and Flow Queries ....................................... 21
      2.2.4 CFL Formulation ...................................................... 24
      2.2.5 Examples ............................................................... 24
      2.2.6 Algorithm ............................................................... 27
      2.2.7 Understanding Self-loops and Improved Algorithm ................. 30
      2.2.8 Demand-driven Algorithm ......................................... 36
   2.3 Soundness ............................................................. 36
   2.4 Discussion and Related Work ........................................... 37
      2.4.1 Polymorphic Subtyping ................................................ 37
      2.4.2 Higher-Order Context-Sensitivity ..................................... 37
      2.4.3 Precise Interprocedural Dataflow Analysis .......................... 39
      2.4.4 Summarization in Context-Sensitive Analyses ....................... 40
      2.4.5 Constraint Simplification ........................................... 40
      2.4.6 Other Related Work .................................................. 41
3 Flow and Type-Structure Polymorphism

3.1 CFL-Based Approach .............................................. 43
   3.1.1 Type Rules .................................................. 43
   3.1.2 Flow Queries in the Presence of Type Polymorphism .... 45
   3.1.3 Practical Aspects of Constraint Generation ............. 47
3.2 POLYEQ: An Efficient Special Case ............................. 48
   3.2.1 Type System ............................................... 49
   3.2.2 Flow Computation ......................................... 49
   3.2.3 Practical Applications ..................................... 50
3.3 Extensions ......................................................... 52
   3.3.1 Non-Structural Subtyping ................................ 52
   3.3.2 Recursive Types ........................................... 54
3.4 Related Work ...................................................... 55

4 Conclusions ......................................................... 56

A Soundness Proof .................................................... 60
   A.1 Overview of the proof ....................................... 60
   A.2 Normal Instantiation Contexts .............................. 62
   A.3 Closure Under Substitution .................................. 65
   A.4 Normal Derivations ........................................... 68
   A.5 Existence Of Normal Derivations ........................... 71
   A.6 Soundness Theorem ........................................... 76
Chapter 1

Introduction

Answering queries of the form “Does any value appearing at program point \( \ell_1 \) flow to program point \( \ell_2 \)” solves many static analysis problems such as finding potential pointer aliases, determining possible targets of indirect function calls, and delimiting storage escape paths. This article studies efficient techniques to answer such queries in a context-sensitive manner, i.e., without introducing spurious flow between different calling contexts.

From Types to Flow

We study the flow of program values in terms of flow paths on the type structure of a program. To make this more concrete, assume that our program \( p \) is given such that each subexpression \( e \) is labeled with a label \( \ell \) as in \( e^\ell \). The queries we can ask of such a system are all queries of the form “Is there flow from \( \ell_1 \) to \( \ell_2 \)?”, where \( \ell_1 \) annotates some subexpression \( e_1 \) and \( \ell_2 \) annotates some subexpression \( e_2 \) of the entire program expression \( e \). We answer such queries by looking at the type \( \tau_1 \) and \( \tau_2 \) inferred for expressions \( e_1 \) and \( e_2 \). Types in our system themselves have labels annotating all subcomponents, thus we find that some label \( \ell_1^\tau \) annotates \( \tau_1 \) and some label \( \ell_2^\tau \) annotates \( \tau_2 \). We then ask whether there is flow from \( \ell_1^\tau \) to \( \ell_2^\tau \) in a type instantiation graph that results from the type inference on \( p \).

The choice of answering flow queries on the type structure of a program is quite natural since sound type systems compute a conservative approximation of value flow. Without subtyping nor polymorphism, this view leads to the following symmetric flow relation: If \( e_1 \) and \( e_2 \) have the same type, then there is potentially flow from \( e_1 \) to \( e_2 \), and vice versa. If their types differ, we know there is no flow.

Using standard types for flow leads to rather unprecise results. To improve the precision, we label type constructors in order to distinguish unrelated occurrences of the same type. For example, the distinct labeling of the argument and result type in the function type \( \text{int}^{\ell_1} \rightarrow \text{int}^{\ell_2} \) indicates that the result is unrelated to the argument.

Another improvement in precision is obtained by treating flow as an asymmetric relation through the introduction of subtyping. Given a function \( \text{int}^{\ell_1} \rightarrow \text{int}^{\ell_2} \) and an argument \( \text{int}^{\ell_0} \), standard language semantics state there is flow from the argument to the function domain, not vice versa. In subtyping, this fact is expressed by requiring that the argument type is a subtype of the domain type \( \text{int}^{\ell_0} \leq \text{int}^{\ell_1} \), which in turn is satisfied if \( \ell_0 \leq \ell_1 \). The flow constraint \( \ell_0 \leq \ell_1 \) now expresses that values arising at expressions characterized by types labeled \( \ell_0 \) flow to expressions with types labeled by \( \ell_1 \).
Precision is further improved by considering a context-sensitive flow analysis based on polymorphic types. The focus of this article is the interaction between subtyping and polymorphism.

Contributions

The main contributions of this article are:

- A novel algorithm for computing context-sensitive, directional flow information for higher order typed programs. Our algorithm improves the asymptotic complexity of the best known algorithm [Mos96] based on polymorphic subtyping from $O(n^8)$ to $O(n^3)$.

- Our results are founded on a novel analysis of subtyping combined with instantiation (semi-unification) constraints including polymorphic recursion. Instantiation constraints provide a graph-based interpretation of substitution, and the ensuing formulation of polymorphic subtyping via CFL-reachability leads to demand-driven and purely graph-based implementation technology for type-based flow analysis.

- We transfer results on precise interprocedural dataflow analysis based on CFL-reachability [RHS95] to the setting of type-based analysis, resulting in an algorithm which works directly on higher order programs with structured data.

- We also study the applicability of CFL-reachability for flow computation in the case where the underlying type structure of programs is itself polymorphic (and not just the labeling), as e.g., in ML-like languages. We show that no additional approximations are necessary in this case and that flow queries are still computable in time $O(n^3)$.

- Our results open the door to new implementation techniques and engineering trade-offs for flow analyses based on polymorphic subtyping systems. By obviating the need to simplify and copy systems of subtyping constraints, our technique may circumvent one of the main scaling inhibitors for such systems.

Many context-sensitive flow analyses based on function summaries (e.g. [CRL99, LH99, FEA99]) are presented as two phase computations. In phase 0, information is propagated from the callees (where it originates) to the callers. In phase 1, information is propagated from callers back to callees. The information in the second step represents summary information for a callee from all contexts. In previous work, this phase distinction does not encompass first-class functions (or function pointers), because uses of functions do not necessarily coincide with call sites in the presence of first-class functions. Our results show that the phase distinction generalizes to first-class functions and is present on each individual flow path, thus enabling fine-grained demand-driven algorithms without the need for two global phases.

An Example

We now consider a very simple example program to show the principal difference of our work to previous approaches. We assume here that the underlying types, which arise by erasing all labels, are monomorphic but the type system is polymorphic over labels. Polymorphism is introduced at let- and letrec-bindings. Occurrences of let- or letrec-bound symbols are uniquely identified by indices $i$, as in $f^i$. Indices $i$ thereby single out individual instantiation
sites. Our framework is similar to the one studied in, e.g., [Mos96]. Detailed definitions are given in Section 1.6.

Our example program e is as follows:

\[
\begin{align*}
\textbf{let } id &= \lambda x : \text{int}^{\ell_1} . x^{\ell_2} \\
\textbf{in } &((id^i \circ \epsilon_\ell)^{\ell_4}, (id^i 1^{\ell_5})^{\ell_6}) \\
\end{align*}
\]

end

Here, we are interested in tracking the flow of constants 0 and 1 labeled with \(\ell_3\) and \(\ell_5\).

1.1 Constraint Copying Methods

All previous work in polymorphic subtype inference is based on \textit{qualified polymorphic types} of the form \(\forall \ell . C \Rightarrow \sigma\), where \(C\) is a set of captured subtyping constraints qualifying the type \(\sigma\). Since \(C\) may contain quantified labels from \(\ell\), such an approach gives rise to copies of the captured constraints at all instantiation sites for that type. A standard \(^1\) polymorphic constrained type (see, e.g., [Sm94, TS96, Mos96]) for \(id\) is \(\forall \ell_1 \ell_2 . \{\ell_1 \leq \ell_2\} \Rightarrow \text{int}^{\ell_1} \rightarrow \text{int}^{\ell_2}\).

In a \textit{copy-based framework}, program \(e\) is typed by copying the constraint set \(\{\ell_1 \leq \ell_2\}\) associated with \(id\) at each of the instantiation sites \(id^i\) and \(id^j\), yielding the standard polymorphic typing judgment

\[
\{\ell_3 \leq \ell_4, \ell_5 \leq \ell_6\}; \emptyset \vdash e : \sigma
\]

with \(\sigma = \text{int}^{\ell_4} \times \text{int}^{\ell_6}\). Such a typing has four components, from left to right: a set of subtype (or flow) constraints, a type environment (here empty), a term and a type. From this typing we conclude that the value 0 (\(\ell_3\)) flows to the first component of the resulting pair (\(\ell_4\)), and the value 1 (\(\ell_5\)) flows to the second (\(\ell_6\)). Polymorphism over labels, here implemented via constraint copying, keeps the two instantiation sites apart, matching up a call site \((e.g., id^i 0^{\ell_5})\) with its proper return (\(\ell_4\)). A monomorphic analysis, in contrast, typically predicts, imprecisely, that either value (\(\ell_3\) or \(\ell_5\)) flows to either return point (\(\ell_4\) or \(\ell_6\)).

The seeming need to copy subtype constraint sets at every distinct instantiation site has been identified as a major problem, making it very difficult to scale polymorphic subtyping to large programs. \(^2\) The problem has generated a significant amount of research, including work on constraint simplification, which aims at compacting constraint sets before they are copied [FM89, Cur90, Kae92, Sm94, EST95, Pot96, TS96, FA96, AWP97, Reh97, FF97]. It is unlikely that constraint simplification techniques alone will solve this problem, and complete simplification is a hard problem itself [Reh98, FF97].

1.2 A New Method Based on Instantiation Constraints

In this article, we tackle the constraint copying problem in a new way. Our starting point is a new presentation of polymorphic subtyping, based on \textit{instantiation constraints}, which we

\(^1\)Function \(id\) can be given a most general typing without any subtyping constraints, but we choose the present typing for illustrative purposes.

\(^2\)The small size of the constraint set in our toy example is illusory in practice, of course.
call POLYFLOW_{CFL}. Instead of constrained types $\forall i. C \Rightarrow \sigma$, POLYFLOW_{CFL} uses standard quantified types of the form $\forall i. \sigma$, which are given meaning in combination with a global set of constraints. Expression $e$ from the example receives a typing of the form

$$\{ \ell_1 \preceq_{\downarrow} \ell_3, \ell_2 \preceq_{\uparrow} \ell_4, \ell_1 \preceq_{\downarrow} \ell_5, \ell_2 \preceq_{\uparrow} \ell_6 \}; \emptyset \vdash e : \sigma \quad (1.2)$$

A typing now has five components, from left to right: a set of instantiation constraints (a.k.a. semi-unification constraints [Hen93]), a set of flow constraints, a type environment (here empty), a term and a type.

An instantiation constraint $\ell \preceq_{p} \ell'$, with $p \in \{+, \mp\}$, expresses that $\ell'$ is an instance of $\ell$, at instantiation site $i$. The indices $i$ and $j$ serve to keep distinct instantiation sites apart. The polarities $p \in \{+, \mp\}$ are explained in a moment. The instantiation constraints of our typing explicitly represent the label substitutions $\varphi_i = \{ \ell_1 \mapsto \ell_3, \ell_2 \mapsto \ell_4 \}$ and $\varphi_j = \{ \ell_1 \mapsto \ell_5, \ell_2 \mapsto \ell_6 \}$ used at instantiation sites $i$ and $j$, respectively, to produce instances of id's type int$^\ell_i \rightarrow$ int$^\ell_j$. Here, $\varphi_i$ is represented by the constraints $\ell_1 \preceq_{\downarrow} \ell_3, \ell_2 \preceq_{\uparrow} \ell_4$, and $\varphi_j$ is represented by $\ell_1 \preceq_{\downarrow} \ell_5, \ell_2 \preceq_{\uparrow} \ell_6$. In a constraint copying framework, $\varphi_i$ and $\varphi_j$ are applied at sites $i$ and $j$ to copy the subtype constraint set $\{ \ell_1 \leq \ell_2 \}$ associated with id, yielding $\{ \ell_3 \leq \ell_4 \}$ and $\{ \ell_5 \leq \ell_6 \}$.

The crucial difference of POLYFLOW_{CFL} to copy-based systems is that

- Instead of explicitly representing copies of the original constraint system ($\{ \ell_1 \leq \ell_2 \}$), only the substitutions necessary to create them are represented. The constraint copies are thereby implicitly given in terms of the original set ($\{ \ell_1 \leq \ell_2 \}$) and the instantiation constraints.

The flow at all instantiation sites is recoverable in a completely demand-driven fashion through the combination of flow and instantiation constraints. Suppose that we demand to know where $\ell_3$ flows. Drawing flow constraint $\{ \ell_1 \leq \ell_2 \}$ as a directed edge from $\ell_1$ to $\ell_2$ and drawing instantiation constraints $\ell_1 \preceq_{\downarrow} \ell_3, \ell_2 \preceq_{\uparrow} \ell_4$ for site i as dotted edges, we recover the flow from $\ell_3$ to $\ell_4$ at instantiation site $i$ by completing the following diagram, the lower dashed edge representing the "recovered" flow constraint:

$$\begin{array}{c}
\ell_1 \\
\preceq_{\downarrow} \\
\ell_3 \\
\preceq_{\uparrow} \\
\ell_4
\end{array}$$

(1.3)

The diagram gets completed by traveling from $\ell_3$ along the instantiation edge $\ell_1 \preceq_{\downarrow} \ell_3$ in reverse direction, then traveling from $\ell_1$ to $\ell_2$ along the direction of flow (i.e., along $\ell_1 \leq \ell_2$), and then finally traveling from $\ell_2$ to $\ell_4$ along the instantiation edge $\ell_2 \preceq_{\uparrow} \ell_4$. We have thereby recovered the flow that was demanded, using only parts of the constraint systems needed for this query.

A further advantage of instantiation constraints is that all flow is present in the constraint sets $I$ and $C$ obtained in a typing judgment for POLYFLOW_{CFL}. Consider for example the flow of both value 0 ($\ell_3$) and 1 ($\ell_5$) to formal parameter $x$ ($\ell_1$). We recover such flow similarly to the flow above via $\ell_1 \preceq_{\downarrow} \ell_3$ and $\ell_1 \preceq_{\downarrow} \ell_5$. To recover such flow in copy-based systems, the entire typing derivation is required instead of merely the final judgment.

A crucial technical insight of our work is that in order to interpret instantiation constraints as flow constraints, it is necessary to assign them polarities (+ and ⊥) indicating
the direction of flow, in addition to the indices $i$ indicating the instantiation site. A negative ($\cdot$) instantiation edge gets traversed in reverse direction of instantiation, a positive one ($+$) gets traversed along the direction of instantiation. Disregarding polarities and interpreting instantiation constraints as bi-directional flow constraints results in a complete loss of context-sensitivity.

Polarities are assigned to instantiation constraints according to the polarity of the "source types" of instantiations, in our example the negative, resp. positive, occurrences of $\ell_1$, resp. $\ell_2$, in the type $\sigma = \text{int}^{\ell_1} \rightarrow \text{int}^{\ell_2}$ of id.

**CFL-Reachability**

Recovering flow through instantiation and flow constraints is best formulated as **CFL reachability** on a graph formed by flow and instantiation constraints (edges) and labels (nodes). Provided we reverse instantiation edges with negative polarity and label them with opening parentheses ($\langle$, label positive instantiation edges with closing parentheses $\rangle$, and flow edges with $d$, all flow paths spell words from a particular grammar. For example, the path $\ell_3 \xrightarrow{\langle} \ell_1 \xrightarrow{d} \ell_2 \xrightarrow{\rangle}$ spells the word "($d$)" and corresponds to a matched flow path, because the parentheses match up. This formulation rules out spurious flow paths, for instance the path $\ell_3 \xrightarrow{\langle} \ell_1 \xrightarrow{d} \ell_2 \xrightarrow{\rangle} \ell_6$, which corresponds to calling id at instance $i$, but returning at instance $j$.

Matched flow paths bear a close resemblance to the precise interprocedural flow paths of matching call- and return sequences studied in [RHS95] for the case of first order programs manipulating atomic data. Our results in the remainder of this article can be seen as a transfer of flow computation via CFL-reachability to the setting of higher-order and type-based polymorphic systems with structured data.

The transfer of the CFL viewpoint to this setting is non-trivial, because we incorporate higher-order functions, polymorphic recursion, structured data and arbitrary instantiations that may not correspond directly to call-return sequences.

### 1.3 Situating the Analyses

Flow analyses compute information about how values flow from one program point to another. There are many variations of flow computations in the literature. The particular form of flow we are concerned with in this article is characterized by the following flow queries: "Do values arising at expression $e_1$ flow to the program point characterized by expression $e_2$?". There is still a wide choice of precisions for such queries. Particular analyses can be characterized along at least two dimensions: **flow-sensitivity** and **context-sensitivity**. Flow-sensitivity is a slightly misnamed concept referring to whether or not an analysis considers the order of destructive updates. Flow-insensitive analyses ignore the order of updates and are sometimes characterized as considering all interleavings of statements. Flow-sensitive analyses are better characterized as performing strong updates. Modeling strong updates requires must alias information. For purely functional languages, flow-sensitivity is of no concern.

The second dimension—context-sensitivity—characterizes how a flow analysis handles flow paths involving function bodies. Many distinct flow paths may share program points

\[\text{In higher-order programs, a function symbol may occur in contexts that are not call sites}\]
within a function body. Context-insensitive analyses do not distinguish flow information for such program points even if the information arises from different invocation contexts. Context-sensitive analyses keep flow paths involving distinct invocation contexts apart.

Three other dimensions are of interest to characterize the flow analyses considered in this article: whether the analysis is summary-based, higher-order, or directional. Whether a context-sensitive analysis is summary-based or not is a vague concept, but intuitively we consider a flow analysis summary-based, if the analysis examines each function body only once, creating a concise description of the function. This description (or a copy thereof) is employed in different contexts where the function is used. In contrast, non-summary-based approaches perform some degree of reanalysis of a function in new contexts. The important aspect of summary-based approaches is not the exact space-time tradeoff made w.r.t. reanalyzing, but the fact that there exists an explicit description of the effects of a function independently of the analysis algorithm.

We say that an analysis is higher-order, if it works directly on programs with first-class functions (or function pointers) without the need for an explicit call-graph approximation computed through other means. A higher-order flow-analysis computes a call-graph approximation as part of the flow computation by computing the flow of function values.

Finally, directional analyses are characterized by having a non-symmetric notion of flow. Symmetric notions of flow dictate that, whenever there is flow from \( \ell \) to \( \ell' \), there is also flow from \( \ell' \) to \( \ell \). Such notions are often very imprecise, but they may have highly efficient implementations. Monomorphic type systems without subtyping, for example, are symmetric, and type inference in these systems can be implemented by solving equations using fast union-find methods.

The flow analyses studied in this article are summary-based, higher-order, context-sensitive, and directional but flow-insensitive. As we show in Section 3.2, flow-insensitivity does not preclude the use of our techniques on imperative programs.

### 1.4 Context-Sensitivity in Higher-Order Programs

Context sensitivity in first-order programs is defined in terms of valid call-return paths. A path is valid if the call and return edges associated with particular call sites form a well-parenthesized sequence.\(^4\) For higher-order programs (function pointers in C) we must first define what context-sensitivity means for indirect function calls. We explain the context-sensitivity of an analysis in terms of a conceptual copying of function bodies. Consider the example C program below:

```c
typedef int (*FIP)(int *);

int f(int *p) {...}
int g(int *q) {...}

void foo(int a, int b, int c) {
    int ra, rb;
    FIP fp = c?f1:g;
```

\(^4\)With each call site \( i \) is associated a pair of matching parenthesis \( [ \] \). Well-parenthesized sequences are those that can be completed by adding parentheses to either end so as to form a completely parenthesized sequence.
int f_i1(int *p) {...}
int f_i2(int *p) {...}

int g_j1(int *q) {...}
int g_j2(int *q) {...}

void foo(int a, int b, int c) {
    int ra, rb;
    if (c) {
        ra = f_i1(&a);
        rb = f_i2(&b);
    } else {
        ra = g_j1(&a);
        rb = g_j2(&b);
    }
}

Figure 1.1: One expansion per function per call-site

void foo(int a, int b, int c) {
    int ra, rb;
    if (c) {
        ra = f_i1(&a);
        rb = f_i2(&b);
    } else {
        ra = g_j1(&a);
        rb = g_j2(&b);
    }
}

Figure 1.2: One expansion per function occurrence

(1) ra = f_p1(&a);
(2) rb = f_p2(&b);
}

Function pointer \(f_p\) is assigned either function \(f\) at occurrence \(i\) or function \(g\) at occurrence \(j\). Indirect call sites 1 and 2 are using \(f_p\). Consider a polymorphic analysis of this program that treats each indirect call to a particular function independently from other calls. Such an analysis corresponds to analyzing the expanded program shown in Figure 1.1 monomorphically (not context-sensitive). In this expansion, we have two copies \(f_i1\) and \(f_i2\) of function \(f\), one per indirect call site, and similarly for function \(g\). This context-sensitivity is based on functions reaching individual call-sites. It is expensive, since the number of instances of a function depends on the number of indirect call sites it flows to.

Another form of context-sensitivity for higher-order programs is adopted in this article. We only allow one copy of a function per occurrence of a function symbol. In our example, this form corresponds to analyzing the expanded program in Figure 1.2 monomorphically. There is one copy of function \(f\) corresponding to occurrence \(i\) and one copy of function \(g\) corresponding to occurrence \(j\). The same function is called at the two indirect call-sites 1 and 2. This form has the advantage that it generates only as many instances of a function \(f\) as there are occurrences of the symbol \(f\) in the program. On the other hand, the use of fewer instances may lead to less precise results. This form of context-sensitivity corresponds to recursive let-polymorphism [Myc84]. Note that in the first-order case, the two approaches are identical, since function symbols only occur at call sites. The analogy of copying functions is only conceptual (and obviously does not apply in the recursive case).
1.5 Overview

In Chapter 2 we start with the system POLYFLOW which is a flow type system with recursive polymorphism. Recursive polymorphism is used to achieve context-sensitivity, even in the case of recursive invocations of functions. Polymorphism in POLYFLOW is restricted to flow labels. The underlying type structure of the program is monomorphic. In other words, the underlying type system corresponds to a monomorphic version of the ML-type system. For this system we show how a known description [Mos96] based on constraint copying and simplification (POLYFLOW_{cpl}) is equivalent to a system without copying nor simplification based on instantiation constraints and context-free language reachability queries (POLYFLOW_{cfl}). The advantages of POLYFLOW_{cfl} over POLYFLOW_{cpl} include an asymptotically better algorithm answering individual or all queries in $O(n^3)$ instead of $O(n^8)$, where $n$ is the tree size of the type-annotated program (or number of distinct nodes in the type instantiation graph). More importantly, the size of the graph generated prior to answering any queries is linear in $n$. The graph closure is computable on demand for particular queries using the techniques presented by Repe et. al. in [RHS95]. Furthermore, we provide insights into the complexity analysis of CFL-based algorithms through a new CFL algorithm.

Chapter 3 extends the POLYFLOW system to the system POLYTYPE, which is polymorphic in both the flow and the type structure. The underlying typing system of POLYTYPE (without flow annotations) is exactly the Milner-Mycroft [Myc84] type system. We skip the copying system and study directly the system POLYTYPE_{cfl} based on instantiation constraints and CFL-reachability. The novel aspect of POLYTYPE_{cfl} is its combination of two CFL-reachability problems, namely the matching of instantiation sites, as well as the matching of data structure construction and elimination. We show that the two matching problems form a single matching problem by keeping them synchronized at instantiation boundaries. This synchronization is possible because we work with a structural subtyping problem without recursive types. Perhaps surprisingly, the complexity of computing flow for POLYTYPE_{cfl} is no harder than in POLYFLOW_{cfl}: individual or all queries are computable in time $O(n^3)$. The two complexities differ however in how the number of nodes $n$ vary with the size of the program $m$. In the worst case for POLYFLOW_{cfl} $n$ is exponential in $m$, whereas for POLYTYPE_{cfl}, $n$ is doubly exponential in $m$. In practice it makes more sense to compare programs where the largest types are roughly the same size, and in that case, the complexity is also the same. In fact, we expect POLYTYPE_{cfl} to have fewer nodes in practice due to the type polymorphism which avoids expanding types of generic functions.

In Section 3.2 we discuss a special restricted case of POLYTYPE called POLYEQ which is polymorphic over flow and type variables, but without subtyping. POLYEQ uses equivalence classes to describe intra-procedural flow. The flow between function instantiations (inter-procedural flow) is directed and we show that it is a special case of the CFL-reachability of POLYFLOW_{cfl} that has a more efficient implementation. Individual flow queries can be answered in linear time and all queries in quadratic time. We show experimentally that POLYEQ scales well to large programs.

At the end of Chapter 3 we discuss how to combine our approach with non-structural subtyping and recursive types. We show restrictions on non-structural subtyping that supposedly guarantee the existence of flow paths with perfectly nesting parentheses. Combining context-sensitivity with recursive types yields two interleaved CFL problems. Reps has shown this problem to be undecidable [Rep00]. In the type-based formulation, the undecidability manifests in the need to label infinite type with infinitely many distinct labels. In practice, one uses repetitive labelings using only a finite number of labels, thus yielding a
\[(\text{Expressions}) \quad e \ ::= \ x \mid n \mid (e_1, e_2) \mid \lambda x.\! e \mid e_1 e_2 \mid \]
\[\quad \text{let } f : \tau = e_1 \text{ in } e_2 \mid f^i \mid \]
\[\quad \text{letrec } f : \tau = e_1 \text{ in } e_2 \mid \]
\[\quad \text{if0 } e_0 \text{ then } e_1 \text{ else } e_2 \mid e.\! j \]

\[(\text{Types}) \quad \tau \ ::= \ \text{int} \mid \tau \rightarrow \tau \mid \tau \times \tau \]
\[(\text{Labelled Types}) \quad \tau \langle s \rangle, \sigma \ ::= \ \text{int}_\ell \mid \sigma \rightarrow_\ell \sigma \mid \sigma \times_\ell \sigma \]
\[(\text{Labels}) \quad L \ ::= \ \ell \mid \]
\[(\text{Label Sequence}) \quad s, t \ ::= \ \ell \mid s \mid s \]

Figure 1.3: Definitions

Computable approximation of the general case.

Chapters 2 and 3 are concerned with the logical aspects of the flow systems under study show only how to infer constraints and compute flow queries on the resulting type instantiation graph. We assume that the underlying type structures are inferred through other means. It is obvious that such type structures can in fact be inferred via algorithm \(W\) for Hindley-Milner typings or the semi-algorithm of Henglein [Hen93] for Milner-Mycroft typings. Chapter 4 offers conclusions, and Appendix A provides a proof of soundness for the flow relation of \(\text{POLYFLOW}_{\text{CFL}}\).

1.6 Definitions

Throughout this article, we study the simple ML-like language given in Figure 1.3. The language has constants (integers \(n\)), pairs, abstraction, application, let and recursive bindings, conditionals, and pair selection \(e.\! j\). We distinguish between lambda-bound variables \(x\) and let- or letrec-bound variables \(f\). Uses of let- and letrec-bound variables are annotated with an instantiation site \(i\) as in \(f^i\). Furthermore, we assume type annotations on lambda-, letrec-, and let-bound variables as well as on instances of let- and letrec-bound variables. Each sub-expression in the source is implicitly labeled, as in \(e.\! \ell\). However, we omit that labeling except where it is interesting. Note that letrec-bindings can bind any kind of value. For instance, mutually recursive functions can be defined by letrec-bindings of a pair of functions.

We work with unlabeled types \(\tau\) and labeled types \(\sigma\). As a technical device, we use the notation \(\tau \langle s \rangle\) to denote labeled types with structure \(\tau\). The label sequence \(s\) is the labeling of \(\tau\) obtained as a pre-order traversal of its structure. Figure 1.4 makes the relation between \(\tau \langle s \rangle\) and \(\sigma\) precise. A judgment \(\vdash \tau \langle s \rangle : \sigma\) states that type \(\tau \langle s \rangle\) is well-labeled and equal to \(\sigma\). When \(\sigma\) is not of interest, we will also write \(\vdash \tau \langle s \rangle\) \(\text{wl}\) for such statements. We use well-labeled types \(\tau \langle s \rangle\) interchangeably with \(\sigma\).

Positive and Negative Polarity

We say that \(\tau_0\) has positive polarity in \(\tau\), if 1) \(\tau = \tau_0\), or 2) \(\tau = \tau_1 \rightarrow \tau_2\) and either \(\tau_0\) has positive polarity in \(\tau_2\), or \(\tau_0\) has negative polarity in \(\tau_1\), or 3) \(\tau = \tau_1 \times \tau_2\) and \(\tau_0\) has positive polarity in \(\tau_1\) or \(\tau_2\).

Similarly, \(\tau_0\) has negative polarity in \(\tau\), if 1) \(\tau = \tau_1 \rightarrow \tau_2\) and either \(\tau_0\) has negative polarity in \(\tau_2\) or \(\tau_0\) has positive polarity in \(\tau_1\), or 2) \(\tau = \tau_1 \times \tau_2\), and \(\tau_0\) has negative polarity in \(\tau_1\) or \(\tau_2\).
\[ \vdash \tau(s) : \sigma \]
\[ \vdash \text{int}(\ell) : \text{int}^{\ell} \]
\[ \vdash \tau_1(s_1) : \sigma_1 \quad \vdash \tau_2(s_2) : \sigma_2 \]
\[ \vdash \tau_1 \times \tau_2(\ell s_1 s_2) : \sigma_1 \times^{\ell} \sigma_2 \]
\[ \vdash \tau_1 \rightarrow \tau_2(\ell s_1 s_2) : \sigma_1 \rightarrow^{\ell} \sigma_2 \]

Figure 1.4: Well-labeling

\[ C \vdash L \leq L \]

\[ C, L_1 \leq L_2 \vdash L_1 \leq L_2 \text{[Id]} \]
\[ C \vdash L \leq L \text{[Ref]} \]
\[ C \vdash L_0 \leq L_1 \quad C \vdash L_1 \leq L_2 \quad \text{[Trans]} \]
\[ \vdash C \vdash L_0 \leq L_2 \]

Figure 1.5: Constraint relation

The definition is identical for labeled types \( \sigma \). A label annotating a a type \( \tau_0 \) with positive (negative) polarity within \( \sigma \) itself has positive (negative) polarity. We also call a label with negative polarity within \( \sigma \) an input label, and one with positive polarity an output label. The set of labels with positive polarity in \( \sigma \) are written \( \sigma^+ \) and the set of labels with negative polarity in \( \sigma \) is written \( \sigma^- \).

**Contexts**

We assign each subexpression \( e' \) of an expression \( e \) a context \( Q \) as follows. The context of \( e_1 \) in an expression \( \text{let}^{\ell} f = e_1 \ldots \) is the label \( \ell \) of the let-expression. The context of \( e_1 \) in \( \text{letrec}^{\ell} f = e_1 \ldots \) is the label \( \ell \) of the letrec-expression. In all other cases, the context of an immediate sub-expression \( e' \) of \( e \) is the same as the context of \( e \). The context of the top-level expression is called the top-level context or \( \ell_\top \). When a context is nested within another context, we call it a sub-context of the latter.

**Flow Constraints**

A flow constraint set \( C \) is a set of constraints of the form \( L \leq L \). Statements of the form \( C \vdash L \leq L' \) express that \( C \) implies \( L \leq L' \). These statements are governed by the rules in Figure 1.5. The generalized form \( C \vdash C' \) abbreviates \( C \vdash L \leq L' \) for each \( L \leq L' \) in \( C' \).

**Structural Subtyping**

Subtyping judgments relate two labeled types and have the form \( C \vdash \sigma \leq \sigma' \). In this article we only consider structural subtyping, i.e., \( \sigma \) and \( \sigma' \) have the same structure \( \tau \). The
$$C \vdash \sigma \leq \sigma$$

$$\frac{C \vdash \ell_1 \leq \ell_2}{C \vdash \text{int}^\ell_1 \leq \text{int}^\ell_2} \quad \text{[Int]}$$

$$\frac{C \vdash \sigma_1 \leq \sigma_1' \quad C \vdash \sigma_2 \leq \sigma_2' \quad C \vdash \ell \leq \ell'}{C \vdash \sigma_1 \times^\ell \sigma_2 \leq \sigma_1' \times^\ell \sigma_2'} \quad \text{[Pair]}$$

$$\frac{C \vdash \sigma_1 \leq \sigma_1' \quad C \vdash \sigma_2 \leq \sigma_2' \quad C \vdash \ell \leq \ell'}{C \vdash \sigma_1 \rightarrow^\ell \sigma_2 \leq \sigma_1' \rightarrow^\ell \sigma_2'} \quad \text{[Fun]}$$

Figure 1.6: Subtype relation

Subtyping is restricted to the labeling as given by the rules in Figure 1.6. As is standard, the function type constructor is contravariant in the domain.

Substitutions

Substitutions $\varphi$ map labels to labels. The domain of a substitution $\varphi$ is the set of labels on which $\varphi$ is not the identity. Explicit simultaneous substitutions replacing $a$ for $b$ and $c$ for $d$ are written $[a/b, c/d]$. We apply explicit substitutions in postfix form, as in $\sigma[\ell_1/\ell_2]$, and named substitutions in prefix form $\varphi(\sigma)$. Substitutions applied to constraint sets $C$ are defined naturally.

Instantiation Constraints

An instantiation constraint $L \preceq^i_p L'$ states that $L$ instantiates to $L'$ at site $i$ with polarity $p$. We use instantiation constraints to make substitutions at instantiation sites of polymorphic types explicit. A polarity $p$ is either positive $+$ or negative $\div$. The operator $\overline{p}$ negates the polarity of $p$. Contrary to subtyping systems, instantiation constraints are covariant in all constructors. We introduce polarities to encode the usual contravariance as an extra annotation. Sets of instantiation constraints are written $I$. Figure 1.7 lifts instantiations to labeled types.

$$I \vdash \sigma \preceq^i_p \sigma \quad I \vdash L \preceq^i_p L$$

$$\frac{I, L \preceq^i_p L' \vdash L \preceq^i_p L'}{I, L \preceq^i_p L' \vdash L \preceq^i_p L'} \quad \text{[Id]}$$

$$\frac{I \vdash \ell \preceq^i_p \ell'}{I \vdash \text{int}^\ell \preceq^i_p \text{int}^\ell'} \quad \text{[Int]}$$

$$\frac{I \vdash \ell \preceq^i_p \ell' \quad I \vdash \sigma_1 \preceq^i_p \sigma_1' \quad I \vdash \sigma_2 \preceq^i_p \sigma_2'}{I \vdash \sigma_1 \times^\ell \sigma_2 \preceq^i_p \sigma_1' \times^\ell \sigma_2'} \quad \text{[Pair]}$$

$$\frac{I \vdash \ell \preceq^i_p \ell' \quad I \vdash \sigma_1 \preceq^i_p \sigma_1' \quad I \vdash \sigma_2 \preceq^i_p \sigma_2'}{I \vdash \sigma_1 \rightarrow^\ell \sigma_2 \preceq^i_p \sigma_1' \rightarrow^\ell \sigma_2'} \quad \text{[Fun]}$$

Figure 1.7: Instantiation Relation
Sets of instantiation constraints $I$ are subject to the well-formedness condition that for any particular index $i$ and any label $L$, there exists at most one $L'$ such that there is a constraint $L \preceq^i_p L'$ in $I$. A set $I$ of instantiation constraints thus gives rise to a vector of substitutions $\Phi(I)$, indexed by instantiation site $i$, where

$$
\Phi_i(L) = L' \text{ if } L \preceq^i_p L' \in I
$$

We use the notation $I \vdash \sigma \preceq^i_p \sigma' : \varphi$ to mean that the instantiation is derivable under $I \vdash \sigma \preceq^i_p \sigma'$, and that $\varphi(\sigma) = \sigma'$. Note that $\varphi$ only has to agree with $\Phi_i(I)$ on the labels appearing in $\sigma$. 

14
Chapter 2

Flow Polymorphism

In this Chapter we study POLYFLOW, a simple flow type system with polymorphic flow variables, but no polymorphism of type structure. We give two presentations of POLYFLOW. We first examine POLYFLOW$_{\text{copy}}$ based on copying constraints at instantiations. Second, we present the system POLYFLOW$_{\text{CFL}}$ based on CFL-reachability. We show that both systems compute equivalent flow information. The advantage of POLYFLOW$_{\text{CFL}}$ is its asymptotically cheaper running time of $O(n^3)$ vs. $O(n^8)$ for the POLYFLOW$_{\text{copy}}$ system.

2.1 A Copy-based System

POLYFLOW$_{\text{copy}}$ is similar to the system Møssin studies in his thesis ([Mos96], Chapter 5). It differs however in the following points.

- Møssin’s approach can only answers queries of the form “What construction points flow to any particular program point?”, where construction points are lambda, pair, and constant expressions. It is inherent in his system that the construction points be identified prior to the analysis. POLYFLOW$_{\text{copy}}$ studied here is slightly more general in that it allows all queries of the form “Is there flow from program point $\ell_1$ to $\ell_2$?”. As a result, we do not require a class of constant labels.

- We make instantiation substitutions explicit via instantiation constraints.

2.1.1 Polymorphic Constrained Types

Polymorphic types for POLYFLOW$_{\text{copy}}$ are qualified by a set of constraints, written $\forall \vec{i}, C \Rightarrow \sigma$. Such a type scheme can be instantiated to $\sigma[\vec{\ell_i}/\vec{i}]$ in all contexts where the constraints $C[\vec{\ell_i}/\vec{i}]$ hold.

The captured constraints $C$ of a polymorphic constrained type describe the flow paths between input labels (labels annotating negative occurrences of types) and output labels (labels annotating positive occurrences of types) in $\sigma$. These constraints summarize all flow between input and output labels of $\sigma$.

We use the notation $\mathit{fl}(\sigma)$ for the set of labels in $\sigma$ and $\mathit{fl}(A)$ for the set of free labels in the environment $A$. We may also use this notation on flow constraints $\mathit{fl}(C)$.
2.1.2 Type Rules

Figure 2.1 contains typing rules for deriving judgments of the form $C_\varphi; I; C; A \vdash e : \sigma$, where $C_\varphi$ is an accumulation of all flow constraints appearing in the derivation, $I$ is a set of instantiation constraints, $C$ a set of flow constraints, $A$ a type environment mapping lambda-bound variables $x$ to labeled types $\sigma$, and let- and letrec-bound variables $f$ to type schemes. The rules are classified into base rules and polymorphic rules. The base rules are mostly standard. Rule [Lam] uses the type annotation $\tau$ to obtain the labeled type $\tau(s)$ used in the judgment of the lambda body. Rule [Label] connects the label $\ell$ annotating an expression $e^\ell$ with the label $\ell'$ annotating the top-level constructor of the type $\tau(\ell's)$ of $e$.

The statement $C \vdash \ell = \ell'$ is an abbreviation for the statements $C \vdash \ell \leq \ell'$ and $C \vdash \ell' \leq \ell$. Tying the expression label $\ell$ to the type label $\ell'$ in such a way allows us to ask our flow-queries on the expression labels, but answer them using the constraints used in the type derivation.

The polymorphic rules involve quantification and instantiation of labels. Rule [Let] differs from standard let-rules in that the set of constraints $C'$ required to derive $e_1$'s type is captured in the polymorphic type $\forall \ell C' \Rightarrow \sigma_1$. The set of quantified labels $\ell'$ contains all free labels of $\sigma_1$ and $C'$ that are not free in $A$ or $C$. Furthermore, $C'$ must be a subset of the collection $C_\varphi$. Rule [Inst] gives types to occurrences $f^\ell$ of let- and letrec-bound variables. The polymorphic type $\forall \ell C' \Rightarrow \sigma$ is instantiated to $\sigma'$ via a substitution $\varphi$ on the quantified labels $\ell$. The judgment $I \vdash \sigma \leq^1 \sigma': \varphi$ requires that the substitution of labels in $\sigma$ is explicit as a set of instantiation constraints in $I$. Recall that this judgment only relates $I$ and $\varphi$ for labels actually occurring in $\sigma$. Furthermore, we require that the constraints $C$ of the context imply the constraints $\varphi(C')$ as expressed by the judgment $C \vdash \varphi(C')$. Given the definition of the relation $C \vdash \varphi(C')$, we note that except for trivial and transitive constraints, $C$ must contain all of the constraints $\varphi(C')$. This aspect gives rise to the copying of constraints: the constraints $C'$ representing the flow paths of the let- or letrec-bound expression are duplicated at each instance by requiring $\varphi(C')$ to be part of the constraints $C$.

Rule [Rec] is used to type letrec-expressions. The first line in the antecedent handles the proper fixpoint requirements, namely that we derive type $\sigma_1$ for $e_1$ under constraints $C'$ and the assumption that $f$'s type is $\forall \ell C' \Rightarrow \sigma_1$. As in the [Let] rule, we require that $C'$ is a subset of the collection $C_\varphi$.

**Contexts of Labels**

We assume that in a derivation of a judgment $C_\varphi; I; C; A \vdash e : \sigma$, all labels of sub-expressions of $e$ are distinct, and that quantified labels of distinct quantified types are distinct via renaming of bound labels. Recall from Section 1.6 that let- and letrec-bindings divide the sub-expressions of $e$ into a set of disjoint contexts. We associate each label $\ell$ appearing in the derivation with a unique context as follows. The context of $\ell$ is the let- or letrec-binding where the label is quantified, or the top-level context, if the label is not bound in any quantified type. Contexts divide the set of labels into equivalence classes $[\ell]$ and furthermore, there is a unique constraint set $C'$ associated with each context, namely the constraint set $C'$ captured in the quantified type of a given let- or letrec-binding. We can thus refer to contexts either via a label (it's context), a let- or letrec-binding, or a particular constraints set $C'$ appearing in the derivation. We use the notation $C_{[\ell]}$ to refer to the constraint system $C'$ associated with the same context as $\ell$.

To give the reader some intuition about contexts, suppose all let- and letrec-bindings generalize lambda expressions, and all lambda’s are let- or letrec-bound, then contexts
\[ C_g; I; C; A \vdash_{cp} e : \sigma \]

**Base Rules**

\[
\frac{}{C_g; I; C; A \vdash_{cp} x : \sigma}[^{[\text{Id}]}] \\
\frac{}{C_g; I; C; A \vdash_{cp} n^\ell : \text{int}^\ell}[^{[\text{Int}]}] \\
\frac{}{C_g; I; C; A \vdash_{cp} e_1 : \sigma_1, e_2 : \sigma_2}[^{[\text{App}]}] \\
\frac{}{\vdash \tau(\lambda \sigma; x : \sigma \rightarrow^\ell \sigma) : C_g; I; C; A \vdash_{cp} \lambda x : \tau. e : \sigma \rightarrow^\ell \sigma}[^{[\text{Lam}]}] \\
\frac{}{C_g; I; C; A \vdash_{cp} (e_1, e_2) : \sigma_1 \times^\ell \sigma_2}[^{[\text{Pair}]}] \\
\frac{}{\vdash \tau(\ell^j s) : C_g; I; C; A \vdash_{cp} e : \sigma_1 x^\ell \sigma_2}[^{[\text{Proj} j = 1, 2]}] \\
\frac{}{C_g; I; C; A \vdash_{cp} \text{if } e_0 \text{ then } e_1 \text{ else } e_2 : \sigma}[^{[\text{Cond}]}] \\
\frac{}{C_g; I; C; A \vdash_{cp} e : \sigma}[^{[\text{Sub}]}] \\
\frac{}{C_g; I; C; A \vdash_{cp} \ell : \text{gen}(A, \sigma_1, C')}[^{[\text{Label}]}] \\
\]

**Polymorphic Rules**

\[
\frac{}{C_g; I; C'; A \vdash_{cp} e_1 : \sigma_1}[^{[\text{Let}]}] \\
\frac{}{C_g; I; C'; A, f : \forall \ell. C' \Rightarrow \sigma_1 \vdash_{cp} e_2 : \sigma_2}[^{[\text{Let}]}] \\
\frac{C' \sqsubseteq C_g, \ell \sqsubseteq \text{gen}(A, \sigma_1, C')}{}{C_g; I; C; A \vdash_{cp} \text{let } f = e_1 \text{ in } e_2 : \sigma_2}[^{[\text{Rec}]}] \\
\frac{}{C_g; I; C'; A \vdash_{cp} \text{letrec } f : \tau = e_1 \text{ in } e_2 : \sigma_2}[^{[\text{Rec}]}] \\
\frac{}{I \vdash \varphi \subseteq^i \sigma' : \varphi \vdash \varphi(C') \text{ dom } \varphi = \ell}[^{[\text{Inst}]}] \\
\]

\[ \text{gen}(A, \sigma, C') = \text{fl}(\sigma, C') \setminus \text{fl}(A) \]

**Figure 2.1**: Copy-Based System POLYFLOW\textsubscript{copy}
coincide with lambda bodies.

2.1.3 Flow Graphs

A flow graph $G = (I, C_y, L)$ of a derivation $C_y; I : C; A \vdash e : \sigma$ is formed by the set of labels $L$ appearing in the derivation together with the set of instantiation edges $I$ and the collection of flow constraints $C_y$.

Figure 2.2 shows an example flow graph. Solid edges represent flow constraints, dotted edges represent instantiation constraints. Large circles group nodes of the same context. Instantiation edges usually cross contexts, but recursive instantiations may produce instantiation edges within a single context. Flow edges usually stay within a context, but can cross between a context and one of its sub-contexts. These cross-context flow edges arise through the use of lambda-bound variables in inner contexts.

Answering a flow query of the form “Is there flow from $\ell_1$ to $\ell_2$” requires finding a path in a flow graph from $\ell_1$ to $\ell_2$ involving flow and instantiation edges. In the special case where $\ell_1$ and $\ell_2$ belong to the same context, the query can be answered by inspecting the flow constraints of the common context $C_{[\ell_1]}$ alone. This property is due to the copying of the bound constraints at instantiations of quantified types.

The next section makes answering flow queries precise using a simple flow relation.
\[
I; C_g \vdash_{cp} \ell \leadsto \ell
\]

\[
\frac{C_g \vdash \ell_1 \leq \ell_2}{I; C_g \vdash \ell_1 \leadsto_p \ell_2} \text{[Level]}
\]

\[
\frac{I, \ell_1 \preceq^+ \ell_2; C_g \vdash \ell_1 \leadsto^+ \ell_2}{[Out]}
\]

\[
\frac{I, \ell_1 \preceq^\leq \ell_2; C_g \vdash \ell_2 \leadsto^\leq \ell_1}{[In]}
\]

\[
\frac{I; C_g \vdash \ell_0 \leadsto_p \ell_1 \quad I; C_g \vdash \ell_1 \leadsto_p \ell_2}{I; C_g \vdash \ell_0 \leadsto_p \ell_2} \text{[Trans]}
\]

\[
\frac{I; C_g \vdash \ell_0 \leadsto^+ \ell_1 \quad I; C_g \vdash \ell_1 \leadsto^\leq \ell_2}{[Stage]}
\]

Figure 2.3: Flow relation for POLYFLOW_copy

2.1.4 Flow Relation

Given a derivation \( C_g; I; C; A \vdash_{cp} e : \sigma \), we can answer flow queries pertaining to labels \( L(e) \) of \( e \) by considering the constraint sets \( I \) and \( C_g \). Figure 2.3 contains a set of rules for deriving flow judgments of the form \( I; C_g \vdash_{cp} \ell_1 \leadsto \ell_2 \) stating that under constraints \( I \) and \( C_g \), there is flow from label \( \ell_1 \) to label \( \ell_2 \). We now examine these rules in turn.

Some flow paths only involve flow constraints in \( C_g \). These can be deduced directly from \( C_g \) via rule [Level], i.e., if we can derive \( C_g \vdash \ell_1 \leq \ell_2 \), then we can deduce flow from \( \ell_1 \) to \( \ell_2 \).

Note that the rules internally use two auxiliary judgments of the form \( I; C_g \vdash \ell_1 \leadsto^+ \ell_2 \) and \( I; C_g \vdash \ell_1 \leadsto^\leq \ell_2 \) called positive and negative flow respectively. Rules with an occurrence of \( p \) are actually rule schemes for all rules obtained by selecting \( + \) or \( \leq \) for \( p \).

Rule [Trans] encodes transitivity for both auxiliary relations. The interesting rules are [In] [Out] and [Stage]. These rules capture that all flow paths take on the form of a positive flow, followed by a negative flow, where positive flow involves flow along positive instantiation edges and ordinary flow edges, and negative flow involves flow along negative instantiation edges and ordinary flow edges. The intuition behind the structure of such flow paths can be gleaned from the special case of a first-order program. In first-order programs, function instantiation and calls coincide. Negative instantiation edges represent edges from actual to formal parameters, and positive instantiation edges represent flow of return values to the call site. Due to the copying of constraints, all matching flow through functions (flow involving a matched sequence of call-return edges) is explicit as ordinary flow constraints. The only flow that is not explicit is flow from within a function to some outer context along return edges (positive flow), and back into a different context along argument edges (negative flow). It is exactly this flow that is captured by rule [Stage] of the flow relation.

2.1.5 Example

Figure 2.4 contains an example program and flow graph. Function idpair is the identity on integer pairs. It is instantiated at site \( i \) within \( f \), which in turn is instantiated at site
let idpair = \(\lambda x: \text{int} \times \text{int}. x\) in
let \(f\) = \(\lambda y: \text{int}. \text{id}^i(a^{\ell_i}, b^{\ell_i})\) in
let \(z\) = \((f^j 0).2^j\)...

![Diagram](image)

Figure 2.4: POLYFLOW\(_{c\_p}\) example

\(j\). There are three distinct contexts \(C_0\), \(C_1\), and \(C_2\) corresponding to the body of \(\text{idpair}\),
the body of \(f\), and the remainder of the code. The dashed flow edges in \(C_1\) arise through
broadcasting of the corresponding edges from \(C_0\) at site \(i\). Using the flow relation of Figure 2.3,
we can deduce flow \(I; C_g \vdash_{cp} \ell_b \leadsto \ell_z\) from \(b\) to \(z\), where \(C_g = C_0 \cup C_1 \cup C_2\). The deduction
uses rule [Level] on the dashed edge starting at \(\ell_b\), giving rise to \(I; C_g \vdash \ell_b \leadsto \ell_1\). Combined
via [Trans] with flow \(I; C_g \vdash \ell_1 \leadsto \ell_z\) through [Out] yields \(I; C_g \vdash \ell_b \leadsto \ell_z\). Finally using
[Stage] together with a trivial negative flow \(I; C_g \vdash \ell_z \leadsto \ell_z\) derived from [Level] results in
\(I; C_g \vdash_{cp} \ell_b \leadsto \ell_z\).

2.2 A CFL-based System

We now present the system POLYFLOW\(_c\_\text{FL}\) which computes the same answers to flow queries
as POLYFLOW\(_c\_p\), but does not involve any constraint copying. POLYFLOW\(_c\_\text{FL}\) is conceptually
simpler than POLYFLOW\(_c\_p\) and also more practical. The best known algorithm for
inferring a type derivation for POLYFLOW\(_c\_p\) has complexity \(O(n^3)\) [Moe96] and involves
handling multiple constraint systems, their simplification, and copying. POLYFLOW\(_c\_\text{FL}\) on
the other hand involves only a single flow constraint system. Computing a type derivation
and answering all flow queries can be done in cubic time for POLYFLOW\(_c\_\text{FL}\). We prove the
equivalence of flow queries between POLYFLOW\(_c\_p\) and POLYFLOW\(_c\_\text{FL}\).
2.2.1 Polymorphic Types

POLYFLOW\textsubscript{cfl} uses polymorphic types of the form \( \forall \ell. \sigma \). Only labels are quantified, and unlike for POLYFLOW\textsubscript{copy}, no qualifying constraint systems appear in the polymorphic types.

2.2.2 Type Rules

Judgments for POLYFLOW\textsubscript{cfl} have the form \( t; I; C; A \vdash \text{cfl} e : \sigma \) shown in Figure 2.5. The base rules change minimally \( \text{w.r.t.} \) the copy system. The extra \( t \) component in the judgment is a label sequence containing the labels appearing in all \( \lambda \)-bound types in the environment \( A \). The [Lam] rule makes this explicit by concatenating the label sequence \( s \) of the new \( \lambda \)-bound type \( \tau(s) \) to the current sequence \( t \) for the judgment of the body. The labels of \( \lambda \)-bound variables have a special role in flow computations. Due to nesting of \( \text{let-} \) and \( \text{letrec-} \) definitions within lambda expressions, \( \lambda \)-bound program variables allow the program direct access to non-local scopes. Intuitively\(^1\), such edges skip some call or return edges that are manifest as instantiation edges. In this section we are transforming the flow problem into a CFL-reachability problem by formulating matched call-return flows as a parenthesis matching problem. Cross-context flow edges make it necessary to add extra instantiation edges to recover skipped call or return edges. We will explain the exact details of this aspect of flow paths in more detail later. Suffice it to say here that at \( \text{let-} \) and \( \text{letrec-bindings} \), we capture the current set of free labels as part of the bindings. For \( \text{letrec-} \) and \( \text{let-bound} \) variables, the environment contains pairs of the polymorphic type and a label sequence: \( A, f : (\forall \ell; \sigma, s) \). At instantiation points \( i \) [Inst], we require instantiations \( I \vdash s \leq \_i \_ \) and \( I \vdash s \leq \_i \_ \) which have the effect that for all \( \ell \) in \( s \), there are constraints \( \ell \leq \_i \_ \ell \) and \( \ell \leq \_i \_ \ell \) in \( I \). We will show later how these so-called self-loops enter into the flow computation.

2.2.3 Flow Graphs and Flow Queries

Given a derivation \( \vdash t; C \vdash \text{cfl} e : \sigma \), the flow graph \( G = (I, C, L) \) is defined by the set of labels \( L \) appearing in the derivation, along with the flow edges \( C \) and instantiation edges \( I \).

The flow relation of POLYFLOW\textsubscript{copy} is basically transitivity on the positive and negative flow relations \( \sim_+ \) and \( \sim_- \) involving constraints of \( C_G \). Since we don’t copy any constraints in the POLYFLOW\textsubscript{cfl} system, there are fewer flow constraints in \( C \) than there are in \( C_G \). As a result, we have to use a larger flow relation to capture exactly the same flow as POLYFLOW\textsubscript{copy}. The basic case where POLYFLOW\textsubscript{copy} has an explicit flow \( \ell_1 \sim_\text{p} \ell_2 \) not present in POLYFLOW\textsubscript{cfl} is in the following situation.

\[
\ell_1 \rightarrow \ell_2 \\
| \quad | \\
| \leq_\_ \quad | \leq_+ \\
\ell_1^i \quad \ldots \quad \ell_2^i
\]

POLYFLOW\textsubscript{copy} generates the edge \( \ell_1^i \leq \ell_2^i \) by copying the constraint \( \ell_1 \leq \ell_2 \) at instance \( i \). In POLYFLOW\textsubscript{cfl}, the flow from \( \ell_1^i \) to \( \ell_2^i \) must be discovered through other means. To do so, we introduce an auxiliary flow judgment \( C \vdash \ell_1 \sim_\text{m} \ell_2 \) called matched flow. Matched flow captures flow paths where instantiation edges form matching pairs, negative instantiation

\(^1\)The intuition applies to the first-order case.
\[ t; I; C; A \vdash_{\text{CFL}} e : \sigma \]

**Base Rules**

\[
\frac{}{t; I; C; A, x : \sigma \vdash_{\text{CFL}} x : \sigma} \quad \text{[Id]} \quad \frac{}{t; I; C; A \vdash_{\text{CFL}} \text{int} : \sigma} \quad \text{[Int]} \\
\frac{}{t; I; C; A \vdash_{\text{CFL}} e_1 : \sigma_2 \rightarrow^t \sigma_1 \quad t; I; C; A \vdash_{\text{CFL}} e_2 : \sigma_2}{t; I; C; A \vdash_{\text{CFL}} e_1 e_2 : \sigma_1} \quad \text{[App]} \\
\frac{}{t; I; C; A \vdash_{\text{CFL}} \tau(s) : \sigma \quad t; I; C; A, x : \sigma \vdash_{\text{CFL}} e : \sigma'} \quad \text{[Lam]} \\
\frac{}{t; I; C; A \vdash_{\text{CFL}} \lambda x.t.e : \sigma \rightarrow^t \sigma'} \quad \text{[Pair]} \\
\frac{}{t; I; C; A \vdash_{\text{CFL}} e_1 : \sigma_1 \quad t; I; C; A \vdash_{\text{CFL}} e_2 : \sigma_2}{t; I; C; A \vdash_{\text{CFL}} (e_1, e_2)^t : \sigma_1 \times^t \sigma_2} \\
\frac{}{t; I; C; A \vdash_{\text{CFL}} e : \sigma_1 \times^t \sigma_2 \quad \text{[Proj \hspace{1em} \text{j} = 1, 2]} \\
\frac{}{t; I; C; A \vdash_{\text{CFL}} e : \sigma_1}{t; I; C; A \vdash_{\text{CFL}} \text{if \hspace{1em} e}_0 \hspace{1em} \text{then \hspace{1em} e}_1 \hspace{1em} \text{else \hspace{1em} e}_2 : \sigma} \quad \text{[Cond]} \\
\frac{}{t; I; C; A \vdash_{\text{CFL}} e : \sigma \\ C \vdash \sigma \leq \sigma'} \quad \text{[Sub]} \\
\frac{}{t; I; C; A \vdash_{\text{CFL}} e : \sigma}{t; I; C; A \vdash_{\text{CFL}} \text{let \hspace{1em} f}_1 \hspace{1em} \text{in \hspace{1em} e}_2 : \sigma} \quad \text{[Let]} \\
\frac{}{t; I; C; A, f : (\forall \ell. \sigma_1, t) \vdash_{\text{CFL}} e_2 : \sigma_2}{t; I; C; A \vdash_{\text{CFL}} \text{letrec \hspace{1em} f}_\tau : \sigma_1 \hspace{1em} \text{in \hspace{1em} e}_2 : \sigma} \quad \text{[Rec]} \\
\frac{}{I \vdash \sigma \triangleleft_\ell \sigma' \quad \text{dom} \hspace{1em} \varphi = \ell}{t; I; C; A, f : (\forall \ell. \sigma, t) \vdash_{\text{CFL}} f : \sigma'} \quad \text{[Inst]} \\
\text{gen}(A, \sigma) = fl(\sigma) \setminus fl(A) \\

Figure 2.5: CFL-Based System \text{POLYFLOW}_{\text{CFL}}
\[
I; C \vdash_{\text{CFL}} \ell \leadsto \ell
\]
\[
\frac{C \vdash \ell_1 \leq \ell_2}{I; C \vdash \ell_1 \leadsto_p \ell_2} \quad [\text{Level}]
\]
\[
\frac{I, \ell_1 \preceq \ell_2 ; C \vdash \ell_1 \leadsto_{+} \ell_2}{I, \ell_1 \preceq \ell_2 ; C \vdash \ell_1 \leadsto_{-} \ell_2} \quad [\text{Out}]
\]
\[
\frac{I, \ell_1 \preceq \ell_2 ; C \vdash \ell_1 \leadsto_{-} \ell_1}{I; C \vdash \ell_1 \leadsto_{-} \ell_1} \quad [\text{In}]
\]
\[
\frac{I; C \vdash \ell_0 \leadsto_{p} \ell_1 \quad I; C \vdash \ell_1 \leadsto_{p} \ell_2}{I; C \vdash \ell_0 \leadsto_{p} \ell_2} \quad [\text{Trans}]
\]
\[
\frac{I \vdash \ell_1 \preceq \ell_0 \quad I; C \vdash \ell_1 \leadsto_{m} \ell_2 \quad I \vdash \ell_2 \preceq \ell_3}{I; C \vdash \ell_0 \leadsto \ell_3} \quad [\text{Match}]
\]
\[
\frac{I; C \vdash \ell_0 \leadsto_{+} \ell_1 \quad I; C \vdash \ell_1 \leadsto_{-} \ell_2}{I; C \vdash_{\text{CFL}} \ell_0 \leadsto \ell_2} \quad [\text{Stage}]
\]

\( p = +, -, \cdot, m \)

Figure 2.6: Flow relation for POLYFLOW_{CFL}

edges match up with positive instantiation edges of the same instance. In the above graph, given that we have a trivially matched flow (involving no instantiation edges) from \( \ell_1 \) to \( \ell_2 \), rule [Match] in Figure 2.6 can be used to deduce matched flow from \( \ell_1 \) to \( \ell_2 \), effectively deducing the copied constraint.

Note how the polarity on the instantiation edges defines the flow direction w.r.t. the direction of the instantiation edge. Positive polarity means that flow occurs in the same direction as the instantiation, whereas negative polarity means that flow occurs in the opposite direction of the instantiation.

The complete flow relation \( I; C \vdash_{\text{CFL}} \ell_1 \leadsto \ell_2 \) defined in Figure 2.6 states that under constraints \( I \) and \( C \), there is flow from label \( \ell_1 \) to \( \ell_2 \). The rules differ from the ones for POLYFLOW_{copy} only in the addition of the auxiliary judgment \( I; C \vdash \ell_1 \leadsto_{m} \ell_2 \) for matched flow, and the extra rule [Match]. The sequence of instantiation edges traversed along a matched flow path form a well-parenthesized sequence, where \( \preceq_{\cdot} \) matches \( \preceq_{+} \). Matched flow avoids spurious flow paths that involve negative and positive instantiation edges from distinct instantiations. In the first-order case, such paths correspond to spurious flow from one call site of a function \( f \) to another call site of \( f \). In the general case it corresponds to spurious flow from one instantiation site to another.

The key to formulating flow queries in the POLYFLOW system as a CFL-problem is the tagging of instantiation edges with polarities. Instantiation constraints have been used previously for computing Milner-Mycroft typings [Hen93], software dependencies [OJ97], and dimension inference [Rit95]. However, we are not aware of any previous work of propagating and exploiting polarities on instantiation edges.
2.2.4 CFL Formulation

We now formulate flow queries as a context-free language reachability problem (see for example [MR97]). Given a flow graph $G = (I, C, L)$, construct the graph $G_{\text{CFL}}$ with nodes $L$, and the following labeled edges:

\[
\ell_1 \xrightarrow{p} \ell_2 \quad \text{if } \ell_1 \prec_1 \ell_2 \in I
\]
\[
\ell_1 \xrightarrow{\iota} \ell_2 \quad \text{if } \ell_1 \leq_1 \ell_2 \in I
\]

\[
\ell_2 \xrightarrow{\iota} \ell_1 \quad \text{if } \ell_1 \leq_2 \ell_2 \in I
\]
\[
\ell_1 \xrightarrow{a} \ell_2 \quad \text{if } \ell_1 \leq \ell_2 \in C
\]

Edges with labels $p$ ($n$) correspond to positive (negative) instantiation edges used in the [Out] ([In]) rule of the flow relation. Edges with labels $(i, j)$ correspond to the instantiation edges used in the [Match] rule.

A flow relation $I; C \vdash_{\text{CFL}} \ell_1 \rightarrow \ell_2$ can be derived via rules of Figure 2.6 if and only if there exists a path in $G_{\text{CFL}}$ where the sequence of labels along the path from $\ell_1$ to $\ell_2$ is accepted by the following grammar with start symbol $S$:

\[
S \leftarrow PN
\]
\[
P \leftarrow MP
\]
\[
| pP
\]
\[
| \varepsilon
\]
\[
N \leftarrow MN
\]
\[
| nN
\]
\[
| \varepsilon
\]
\[
M \leftarrow (i, M)_i
\]
\[
| MM
\]
\[
| \varepsilon
\]

It is easy to see that productions for $P$ accept paths that corresponds to positive flow $I; C \vdash \ell_1 \rightarrow \ell_2$, productions for $N$ produce negative flow paths, and productions for $M$ produce matched flow paths.

Note that in practice, the graph $G_{\text{CFL}}$ need not explicitly be computed. Instead, the graph closure can be computed directly on the set of constraints $C$ and $I$.

2.2.5 Examples

Figure 2.7 shows the same example examined for the POLYFLOW_copy case. The top contains the flow graph $G$ containing flow and instantiation edges. The bottom contains the corresponding CFL graph $G_{\text{CFL}}$ where we omit the $d$ labels on flow edges. The flow path from $b$ to $z$ now takes the form

\[
\ell_b \xrightarrow{d} \ell_1 \xrightarrow{d} \ell_z
\]

where the flow from $\ell_b$ to $\ell_1$ forms a matched flow that was explicit as a copied constraint in POLYFLOW_copy.
let idpair = λx:int × int.x in
let f = λy:id (a^i, b^j) in
let z = (f 0), 2^i
...

Flow graph G

CFL Graph G_{CFL}

Figure 2.7: POLYFLOW_{CFL} example
let \( \text{app} = (\lambda f. (\lambda x. f x)^{\ell_f})^{\ell_{\text{app}}} \)
in
let \( \text{id} = (\lambda y. y)^{\ell_\text{id}} \)
in
let \( \text{w} = ((\text{app}^i \, \text{id}^j)^{\ell_{\text{app}}} \, b^i)^{\ell_\text{w}} \)

![Diagram showing flow analysis](image)

**Figure 2.8: Higher-order example (only relevant edges shown)**

One advantage of using the type-based approach to flow analysis presented here is that it deals directly with higher-order programs. Figure 2.8 contains an example. The \text{app} function takes a function \( f \) as an argument and returns a function (labeled \( \ell_f \)) that in turn takes a parameter \( x \) and applies \( f \) to \( x \). The figure shows the flow graph resulting from applying \text{app} at instance \( i \) to the identity function \( \text{id} \) (instance \( j \)) and a value \( b \). We have used boxes around labels annotating function types to make the type structure more readable. First note that we can determine what functions are called indirectly within \text{app}, by observing what labels flow to \( \ell_f \). There is a path

\[
\ell_{\text{id}} \xrightarrow{\ell_8} \ell_9 \xrightarrow{\ell} \ell_f
\]

showing that the identity function (labeled \( \ell_{\text{id}} \)) flows to \( \ell_f \). The flow edges connecting the types labeled by \( \ell_8 \) and \( \ell_9 \) arise from the subtype relation between the instance of \( \text{id} \) (the argument) and the domain of the instance of \text{app}. The reversed edge \( \ell_1 \leq \ell_2 \) arises through contra-variance of the subtype relation for function domains.

Edge \( \ell_2 \leq \ell_6 \) represents the argument passing of \( x \) to \( f \) within \text{app}, and similarly, \( \ell_6 \leq \ell_7 \)
represents the flow of the result of this application to the result of the function labeled \( \ell_r \).

The flow path connecting \( b \) with \( w \) is then as follows:

\[
\ell_6 \rightarrow \bullet \xrightarrow{\ell_x} \ell_x \rightarrow \ell_0 \xrightarrow{i} \ell_1 \rightarrow \ell_2 \xrightarrow{j} \ell_y \rightarrow \ell_3 \xrightarrow{j} \ell_4 \rightarrow \ell_5 \xrightarrow{j} \ell_6 \rightarrow \ell_7 \xrightarrow{j} \bullet \rightarrow \ell_w
\]

Observe how the path enters \( \text{app} \) through instance \( i \) and then emerges back along the edge \( \ell_6 \xrightarrow{i} \ell_1 \). The polarity of this edge was determined to be positive, because the polarity of the argument type \( \ell_f \) within the type of \( \text{app} \) is itself negative. The path then traverses \( \text{id} \) on instance \( j \) and reenters \( \text{app} \) at instance \( i \) through \( \ell_5 \xrightarrow{i} \ell_6 \) before finally emerging along the edge \( \ell_7 \xrightarrow{j} \bullet \). The example shows that in the higher-order case, the traversal of an instantiation edge does not correspond directly to an argument passing or return step as in the first-order case. In this example the path traverses \( \text{app} \) twice through instance \( i \).

### 2.2.6 Algorithm

We now show how to compute flow queries for an expression \( e \) whose size without type annotations is \( m \), and whose type-annotated size is \( n \). This section gives an algorithm with time complexity \( O(n^3) \) for computing individual or all queries. Note that in the worst case, the size \( n \) is exponentially larger than \( m \), although \( n \) will typically be close to \( m \) in practice.

#### Constraint Derivation

The type rules in Figure 2.5 can be used directly as constraint inference rules. As is standard for inference systems, the use of the subsumption rule [Sub] can be restricted to the argument derivation in [App], the branches in [Cond], and the binding in [Rec]. Constraints are inferred at rules [Inst] and [Sub] by using the rules of Figures 1.7 and 1.5 in reverse, i.e., to obtain the conclusion, we need to generate the constraints required by the antecedents. To guarantee the well-formedness of instantiation constraints, the rule presented in [Hen93] must be applied whenever possible:

\[
\ell \leq_p^i \ell_1 \land \ell \leq_p^i \ell_2 \Rightarrow \ell_1 = \ell_2 
\]  

(F1)

This process produces a flow constraint set \( C \) of size \( O(n) \), and an instantiation constraint set of size \( O(mn) \). The \( m \) factor in the size of \( I \) is a direct result of the extra instantiation constraints added on free labels at rule [Inst]. Without these, we would generate only \( O(n) \) instantiation constraints. We show in Section 2.2.7 how to generate only \( O(n) \) constraints and thus obtain a linear sized graph on which to answer flow queries. This space reduction is of practical importance for the demand-driven algorithm we present in Section 2.2.8.

Constraint generation can be implemented in time proportional to the number of derived constraints. The only non-obvious steps are in rules [Let] and [Rec], where we avoid using \( gen(A, \sigma_1) \) to find the quantifiable labels. This problem is solved with an extra subsumption step on \( \sigma_1 \leq \sigma_1' \) guaranteeing that all labels in \( \sigma_1' \) are fresh. Binding this type in place of \( \sigma_1 \) allows instantiation of all labels occurring in \( \sigma_1' \) at all instances. This step also obviates the need to ever apply rule (F1).

#### Answering Flow Queries via CFL Reachability

After constraint derivation we produce the graph \( G_{\text{CFL}} \) according to the description given earlier. This graph has \( O(n) \) nodes (the set of labels in the typed program), and \( O(mn) \)
edges. We follow [MR97] and normalize the grammar for the CFL problem such that the right hand sides of productions contain at most two symbols (terminals or non-terminals), resulting in the grammar shown in Figure 2.9. This grammar has \( m \) terminals \((i, j)\), and \( m \) non-terminals \( K_i \), since the number of distinct instantiations \( i \) is linear in the program size and independent of the type size.

Figure 2.9: Grammar for CFL queries.

\[
\begin{align*}
S & \rightarrow P N \\
P & \rightarrow M P \mid p P \mid e \\
N & \rightarrow M N \mid n N \mid e \\
K_i & \rightarrow (i) K_i \\
M & \rightarrow M M \mid d \mid e \\
\end{align*}
\]

Figure 2.10 shows a generic CFL-algorithm that computes all derivable paths in time \( O(en + un^2 + bn^3) \), where \( e \) is the number of epsilon productions, \( u \) is the number of distinct unary grammar productions of the form \( A \rightarrow B \), and \( b \) is the number of distinct binary grammar productions of the form \( A \rightarrow B C \). Applied to our particular CFL problem (grammar of Figure 2.9) where \( e = 3 \), \( u = 1 \), and \( b = O(m) \), we obtain an initial complexity of \( O(mn^3) \) for computing all pairs reachability. We will further tighten the complexity to \( O(n^3) \) by exploiting the particular structure of constraints generated by POLYFLOW\textsubscript{CFL}.

On termination, the algorithm produces a graph \( G_C \) containing all possible edges labeled by non-terminals of the grammar. To answer a query for flow from \( \ell_1 \) to \( \ell_2 \) we simply inspect \( G_C \) for the presence of an \( S \)-edge from \( \ell_1 \) to \( \ell_2 \).

The algorithm of Figure 2.10 uses a work list \( W \) and adds edges to the result graph \( G_C \). For each node \( \ell \) and each production \( r \) of the form \( A \rightarrow B C \) of the grammar, we use two sets \( pred_r(\ell) \) and \( succ_r(\ell) \) containing the predecessors of \( \ell \) reachable via an edge labeled \( B \), and the successors of \( \ell \) reachable via an edge labeled \( C \).

The algorithm assumes that given an edge labeled \( B \), we can index through \( B \) the rules \( r \) that apply in the for-loops without looking at rules that don’t apply.

We argue the complexity of the algorithm as follows: The number of steps needed to add all edges for epsilon-productions is \( en \). Now consider the statement labeled (2) in the algorithm. For a fixed unary rule \( r \) and node \( \ell_1 \), this statement is executed at most \( n \) times, since the test at (1) guarantees that we see each edge at most once. There are \( u \) rules and \( n \) nodes \( \ell_1 \), thus the overall number of executions of statement (2) is \( un^2 \) times. Next consider the statement labeled (3) in the algorithm dealing with productions of the form \( A \rightarrow B C \). For a particular node \( \ell_2 \) and particular binary production \( r \), this statement is executed at most \( n^2 \) times, because there are at most \( n \) distinct predecessors in \( pred_r(\ell_2) \) and \( n \) distinct successors in \( succ_r(\ell_2) \) that can be paired up. The test at (1) guarantees that we never add a node twice to a bucket \( pred_r \) or \( succ_r \). Since there are \( b \) distinct productions and \( n \) distinct
\[ W = \text{edges of } G_{\text{CFL}} \]
\[ G_0 = \emptyset \]
for each production \( A \rightarrow \epsilon \) and node \( \ell \), add \( \ell \xrightarrow{A} \ell \) to \( W \)
while \( W \) not empty
remove edge \( e = \ell_1 \xrightarrow{B} \ell_2 \) from \( W \)
(1) if \( e \) not in \( G_0 \) do
  add \( e \) to \( G_0 \)
  for each rule \( r \) of the form \( A \rightarrow B \)
    add \( \ell_1 \xrightarrow{A} \ell_2 \) to \( W \)
(2) for each rule \( r \) of the form \( A \rightarrow BC \) do
    add \( \ell_1 \rightarrow \ell_2 \) to \( W \)
    for each \( \ell_3 \) in \( \text{succ}_r(\ell_2) \)
      add \( \ell_1 \xrightarrow{A} \ell_3 \) to \( W \)
  end
end
for each rule \( r \) of the form \( A \rightarrow CB \) do
  add \( \ell_2 \) to \( \text{succ}_r(\ell_1) \)
  for each \( \ell_0 \) in \( \text{pred}_r(\ell_1) \)
    add \( \ell_0 \xrightarrow{A} \ell_2 \) to \( W \)
  end
end
dendif
end

Figure 2.10: CFL algorithm

nodes \( \ell_2 \), we obtain the bound \( O(bn^3) \). The argument for productions of the form \( A \rightarrow CB \)
is analogous. The overall complexity bound of the algorithm is thus \( O(en + un^2 + bn^3) \).

This bound is more precise than the bound of the algorithm presented in [MR97] which has worst case complexity\(^2 \ O(|\Sigma|^3n^3) \). It assumes that the grammar can have \( |\Sigma|^3 \) binary productions and only gives complexity for this case. For this case, our bound is consistent with the bound in [MR97]. But in practice, grammars are smaller and our algorithm attains better bounds in those cases.

**Cubic Algorithm**

The complexity bound of the general algorithm can be tightened by considering \( u \) and \( b \) to be the average number of grammar rules that apply at any particular node. In that case it doesn’t matter what the number of overall distinct grammar rules are. The complexity is solely determined by the average number of rules that apply at each node. The overall complexity improves in the case where \( u \) and \( b \) are constant at each node, but the productions are drawn from a non-constant set of distinct productions (in our case, there are \( O(m) \))

\(^2\Sigma \text{ is the set of terminals and non-terminals used.}\)
distinct productions).

As an example consider the family of $O(m)$ productions of the form $M \leftarrow (\ell_i K_i$. If we
can bound the average number of edges labeled $(\ell_i$ on all nodes in our initial flow graph by
a constant, then on average only a constant number of productions of the form $M \leftarrow (\ell_i K_i$
apply at any node. This hinges on the fact that the algorithm does not add any new edges
labeled with terminals $(\ell_i$.

If we discount the instantiation self-loops of the form $\ell \xrightarrow{\ell_i} \ell$ added through rule $[\text{Inst}]$
for the moment, we obtain the desired bounds on $b$. On average at any label, only a constant
number of productions apply. However, the number of self-loops added through rule $[\text{Inst}]$
is $O(m)$ per label in the worst case. Fortunately, the complexity analysis for general binary
rules given above can be tightened in the case of self-loops. Consider rule $M \leftarrow (\ell_i K_i$, when
applied to a self-loop. The situation is as follows:

$$
\ell \xrightarrow{\ell_i} K_i \xrightarrow{\ell}
$$

For a fixed $\ell$ and fixed $i$, the number of times this rule triggers is at most $n$, since there are at
most $n$ labels $\ell'$ connected to $\ell$ via an edge $K_i$. Since there are at most $m$ such self-loops on
$\ell$ and $n$ distinct labels $\ell$, the number of executions of line 3 in the CFL algorithm involving
self-loops is bounded by $O(mn^2)$. The same argument applies to $K_i \leftarrow M (\ell_i$. This analysis
leads to:

**Theorem 2.1** Given $I$, $C$, $\ell$, $\ell'$. Deciding whether $I; C \vdash_{\text{CFL}} \ell \sim \ell'$ can be done in time
$O(n^3)$. Moreover, the entire flow relation derivable from $I$ and $C$ can be computed in time
$O(n^3)$.

This result improves the best known algorithm, given by Mossin [Mos96], from $O(n^8)$ to
$O(n^3)$. The gain is realized by avoiding repeated copies and simplifications of constraint
sets, and also by avoiding iterating the inference to obtain fixpoints for the polymorphic
recursive type schemes.

2.2.7 Understanding Self-loops and Improved Algorithm

We now show variations of the constraint derivation and the CFL-algorithm such that the
initial set of constraints $I$ generated is of size $O(n)$ instead of $O(mn)$. Recall that through
rule $[\text{Inst}]$, our constraint derivation phase generates up to $O(m)$ distinct instantiation self-
loops of the form $\ell \xrightarrow{\ell_i} \ell$ for up to $O(n)$ distinct labels $\ell$ annotating types of $\lambda$-bound
variables. Consequently, there are up to $O(n)$ nodes that need to consider rules of the form
$M \leftarrow (\ell_i K_i$ for up to $O(m)$ distinct $i$. Thus the average number of rules that may apply per
node is still $O(m)$. To reduce the average to a constant, we need to understand why these
self-loops are needed in the first place.

It is interesting to note that these self-loops $\ell \xrightarrow{\ell_i} \ell$ were used by Henglein [Hen93] merely
as a convenient way of enforcing monomorphism of $\lambda$-bound variables. In POLYFLOW$_{CFL}$,
self-loops are necessary to recover all flow paths and thus for proving soundness of flow.

**Example**

Figure 2.11 shows an example where self-loops are present. The example contains a let-
binding of a function $f$ within the scope of the $\lambda$-bound parameter $x$ of $g$. The body of $f$
let $g = \lambda x: \text{int}^{\ell_x}.$

let $f = \lambda y: \text{int}^{\ell_y}.x$

in

if $0 \ x$ then $(f') \ (0^{\ell_y} \ell_x)$ else $(f') \ (1^{\ell_y} \ell_x)$

in

$(g^h \ 4^{\ell_y} \ell_x)$

Figure 2.11: Self-loop example (p and n edges not shown)

refers directly to $x$ by returning it. Within $g$, we apply two instances of $f$ at sites $i$ and $j$. Finally, $g$ is applied at site $h$. In this example, there is obviously flow from label $\ell_4$ to $\ell_5$, since $g$ acts as the identity function.

The corresponding CFL-graph in Figure 2.11 contains two paths exhibiting the flow from $\ell_4$ to $\ell_5$. The two distinct paths correspond to the two branches of the if-expression, and thus to two distinct applications of $f$, one at $i$, the other at $j$. Each path uses one of the two self-loops on $\ell_x$, namely the one corresponding to the particular instance of $f$ used in the remainder of the path.

$$
\ell_4 \xrightarrow{l_4} \ell_x \xrightarrow{l_4} \ell_x \rightarrow \bullet \xrightarrow{l_4} \ell_2 \rightarrow \bullet \xrightarrow{l_4} \ell_5
$$

$$
\ell_4 \xrightarrow{l_4} \ell_x \xrightarrow{l_4} \ell_x \rightarrow \bullet \xrightarrow{l_4} \ell_3 \rightarrow \bullet \xrightarrow{l_4} \ell_5
$$

31
The self-loops are necessary to balance the result edges $\bullet \xrightarrow{\lambda} \ell_2$ or $\bullet \xrightarrow{\lambda} \ell_3$ from the body of $f$ to the instantiation sites $i$ and $j$. If we think about the closure-converted version of this program, we would explicitly pass $x$ as a parameter to $f$. It is this parameter path that is skipped by referring to free $\lambda$-bound variables like $x$ and the self-loops essentially simulate the extra parameter passing.

**Level Based Inference**

In general, this situation occurs whenever we have a let- or letrec-binding $f$ referring to a $\lambda$-bound variable $x$ from an outer scope and a use of $f$:

$$\lambda x : \sigma. \ldots$$
$$\ldots \text{let } f = \ldots x \ldots$$
$$\text{in}$$
$$\ldots f^i$$
$$\ldots$$

The self-loops on labels $\ell$ in $\sigma$ annotating the type of a $\lambda$-bound variable will have instance labels $i$ for all instances of any let- or letrec-bindings within the lambda body itself. To determine the self-loops on a label $\ell$ annotating a lambda with scope $r$, it is thus sufficient to determine the binding scope $q$ of any instances $i$ within $r$. If $q$ is a sub-scope of $r$, then $\ell$ acquires a self-loop for instance $i$. In fact, it is sufficient to determine the relative syntactic level of $\lambda$, let-, and letrec-bindings instead of the actual scopes. The syntactic level is determined as follows. For a let- or letrec-expression of $f$ occurring at level $k$, its sub-expressions $e_1$ and $e_2$ occur at level $k+1$ and the binding $f$ occurs at level $k$. The following example shows that variables of lambda-bindings occurring within a letrec (or let) binding have a strictly higher level $k+1$ than the letrec bound variable $f$ with level $k$. Thus, no self-loops for labels on $x$ or $y$ are created at instances $i$ or $j$ in this example.

$$\text{letrec } f_k = \lambda x_{k+1}. \ldots f_k^i \ldots$$
$$\text{in}$$
$$\lambda y_{k+1}. \ldots f_k^i \ldots$$

In the case where the let-binding is nested within a lambda-expression, the binding level of $f$ is equal or higher to the binding level $k$ of the lambda bound variable. Thus any instance $i$ of $f$ will generate loops on labels of $x$.

$$\lambda x_k.$$  
$$\text{let } f_k = \ldots$$  
$$\text{in}$$  
$$\ldots f_k^i \ldots$$

These observations can be exploited by introducing summary self-loop edges of the form $\ell \xrightarrow{\ell} \ell$ for each label $\ell$ annotating the type of a lambda-bound variable at level $\ell$. Such a summary self-loop stands for the set of edges with labels $\ell_i$ where $i$ is an instance of a binding of level $q \ge r$ occurring in the scope of the lambda generating $\ell \xrightarrow{\ell} \ell$.

The new rules are shown in Figure 2.12. Judgments no longer contain the sequence of free labels $t$, but are now indexed by the syntactic level $k$: $I; C; A \vdash_{\text{CFL}} e : \sigma$. Furthermore, the environment $A$ maps let- and letrec-bound names to a quantified type paired with the binding level of the name (see rules [Let] and [Rec]). At instantiation of $f^i$ with binding
level \( q \), we require the instantiation constraints \( \sigma \xrightarrow{\ell} \sigma' \) to be annotated with the binding level \( q \).

**Level Based Queries**

In the CFL-graph \( G_{\text{CFL}} \), we label the instantiation edges with \( (i \mid \ell) \) or \( (\ell \mid i) \) \(^3\). Furthermore, for each label \( \ell \) annotating the type of a lambda-bound variable at level \( k \), we add a single summary self-loop \( \ell \xrightarrow{\delta^k} \ell \) to \( G_{\text{CFL}} \). \( G_{\text{CFL}} \) now contains \( O(n) \) nodes and \( O(n) \) initial edges where \( n \) is the tree size of the type-annotated program. The number of summary self-loops is bounded by the number of labels or \( O(n) \).

We rewrite the grammar for the CFL-problem as follows. Since, a summary self-loop \( s^k \) on \( \ell \) replaces the set of self-loops with labels \( \{i \mid \ell\} \) or \( \{\ell \mid i\} \) where \( i \) is an instance of a binding of level \( q \geq k \) occurring in the scope of the lambda generating \( \ell \xrightarrow{\delta^k} \ell \), we need a family of grammar rules of the form

\[
M \leftarrow s^k M \delta^q \quad q \geq k
\]

and

\[
M \leftarrow (i \mid M) s^k \quad q \geq k
\]

which are normalized by introducing the families of auxiliary non-terminals \( L^q \) and \( R^q \).

\[
\begin{align*}
R^q & \leftarrow M \delta^q \\
L^q & \leftarrow \{i \mid M \}
\end{align*}
\]

\[
M \leftarrow s^k R^q \quad q \geq k
\]

\[
\mid L^q s^k \quad q \geq k
\]

The full grammar is shown in Figure 2.13. Note that we still have no more than \( O(m) \) terminals and non-terminals, since the binding level \( k \) of a terminal depends directly on the instance \( i \). But we draw our productions from a family of \( m^3 \) distinct productions.

We now have on average a constant number of edges labeled with terminals in our graph and the average number of distinct rules that apply per node is constant, except for nodes \( \ell \) with summary self-loops \( s^k \), since they need to consider the family of rules \( M \leftarrow s^k R^q \) (and \( M \leftarrow L^q s^k \)) for all \( q \) such that \( q \geq k \). But it is still the case that for a fixed \( \ell \) and \( q \), the number of applications of this rule is bounded by \( O(n) \), since at most \( n \) distinct labels \( \ell' \) are connected to \( \ell \) via an edge labeled \( R^q \). Even though we reduced the number of initial edges to \( O(n) \), the CFL-algorithm of Figure 2.10 would require \( O(m) \) sets \( \succ_r \) at a label \( \ell \) with summary self-loop \( k \), one for each rule with \( q \geq k \). Overall, that leads to \( O(mn) \) such sets. We can modify the algorithm to use only \( O(n) \) such sets, by noting that at any particular label \( \ell \) where the family of rules \( M \leftarrow s^k R^q \) with \( q \geq k \) applies, the set \( \succ_r \) of all these rules can be shared, because it doesn’t matter which particular level \( q \) is used to trigger the rule. This optimization depends on the fact that there are no other rules involving \( R^q \) on the right-hand side. The same reasoning applies to the family \( M \leftarrow L^q s^k \).

To summarize, the introduction of summary self-loops reduces the size of the initial constraint graph from \( O(mn) \) to \( O(n) \) and further allows the CFL-algorithm to use only constant number of \( \text{pred}_r \) and \( \text{succ}_r \) sets per node on average.

\(^3\)One can distinguish positive and negative self-loops that match up only with a negative (positive) instantiation edge. Positively occurring labels only require negative self-loops, negatively occurring labels only require positive self-loops. We chose to elide this detail here.

33
\(I; C; A \vdash \text{CFL}, e : \sigma\)

**Base Rules**

\[
\begin{align*}
I; C; A, x : \sigma & \vdash_{\text{CFL}} x : \sigma & \text{[Id]} \\
I; C; A & \vdash_{\text{CFL}} \text{int}: \text{int} & \text{[Int]} \\
I; C; A & \vdash_{\text{CFL}} e_1 : \sigma_1, e_2 : \sigma_2 & \text{[App]} \\
& \vdash_{\text{CFL}} \tau(s) : \sigma \quad t, s & \vdash_{\text{CFL}} e : \sigma' & \text{[Lam]} \\
I; C; A & \vdash_{\text{CFL}} \lambda x : \tau. e : \sigma \to \ell \sigma' & \text{[Pair]} \\
I; C; A & \vdash_{\text{CFL}} e : \sigma_1 \times \ell \sigma_2 & \text{[Proj]} j = 1, 2 \\
I; C; A & \vdash_{\text{CFL}} e_0 : \text{int} & \text{[Cond]} \\
I; C; A & \vdash_{\text{CFL}} e_1 : \sigma & \text{if } e_0 \text{ then } e_1 \text{ else } e_2 : \sigma \\
I; C; A & \vdash_{\text{CFL}} e : \sigma & \text{C} \vdash \sigma \leq \sigma' & \text{[Sub]} \\
I; C; A & \vdash_{\text{CFL}} e : \sigma' & \vdash_{\text{CFL}} \tau(\ell' s) : \sigma & \vdash \ell = \ell' & \text{[Label]} \\
\end{align*}
\]

**Polymorphic Rules**

\[
\begin{align*}
I; C; A \vdash_{\text{CFL}^{+ \ell}} e_1 : \sigma_1 & \quad \bar{\ell} = \text{gen}(A, \sigma_1) & \text{[Let]} \\
I; C; A, f : (\forall \ell. \sigma_1, k) \vdash_{\text{CFL}^{+ \bar{\ell}}} e_2 : \sigma_2 & \\
I; C; A & \vdash_{\text{CFL}} \text{letrec } f \leftarrow e_1 \text{ in } e_2 : \sigma_2 & \text{[Rec]} \\
I; C; A \vdash_{\text{CFL}} f : e_1 : \sigma_1 & \quad \bar{\ell} = \text{gen}(A, \sigma_1) \\
I; C; A, f : (\forall \ell. \sigma_1, k) \vdash_{\text{CFL}^{+ \bar{\ell}}} e_2 : \sigma_2 & \\
I; C; A & \vdash_{\text{CFL}} \text{letrec } f \leftarrow e_1 \text{ in } e_2 : \sigma_2 & \text{[Rec]} \\
I; C; A & \vdash \sigma \leq_+^q \sigma' : \varphi & \text{dom}(\varphi) = \bar{\ell} & \text{[Inst]} \\
I; C; A, f : (\forall \ell. \sigma, q) \vdash_{\text{CFL}} f^i : \sigma' & \text{gen}(A, \sigma) = fl(\sigma) \setminus fl(A) \\
\end{align*}
\]

Figure 2.12: POLYFLOW_{\text{CFL}} with levels
\[
\begin{align*}
S & \leftarrow P N \\
P & \leftarrow M P \\
& \mid p P \\
N & \leftarrow M N \\
& \mid n N \\
& \mid e \\
N_i & \leftarrow M N_i \\
M & \leftarrow \eta_i N_i \\
& \mid M M \\
& \mid d \\
& \mid e \\
& \mid s^k R^q \quad q \geq k \\
& \mid L^q s^k \quad q \geq k \\
R^q & \leftarrow M N_i^q \\
L^q & \leftarrow \eta_i M
\end{align*}
\]

Figure 2.13: Grammar for summary self-loops

Levels instead of Scopes

By construction, the set of flow paths recognized by the new formulation contains at least all the flow paths of the original formulation. On the other hand, we use only syntactic levels to decide which summary self-loops could pair up with some instance \( i \) instead of actual scopes. It thus appears that a self-loop of a label \( \ell_1 \) on a lambda-bound variable \( y \) at level \( p \) may match up with any instantiation edge \( \ell_2 \xrightarrow{\lambda_i} \ell_3 \) with a binding level \( q \) greater or equal to \( p \), even though the instance \( i \) does not appear in the scope of \( \lambda y \) (see Figure 2.14).

But for such a situation to arise, there must be an edge \( \ell_1 \xrightarrow{R^q} \ell_3 \) which in turn requires a matched path connecting label \( \ell_1 \) with the instantiation edge \( \eta_i \) at label \( \ell_2 \). Such a matched path must go through an outer lambda \( \lambda x \) at level \( k \leq p \) with label \( \ell_0 \), where \( \lambda x \) contains both \( \lambda y \) and the instantiation \( i \) in its scope (refer to Figure 2.14). The label \( \ell_0 \) will have a self-loop that can properly match up with the instantiation edge \( \eta_i \) producing a direct edge \( \ell_0 \xrightarrow{M} \ell_3 \). Thus by transitivity of \( M \)-edges the matched path from \( \ell_1 \) to \( \ell_0 \) combined with the path from \( \ell_0 \) to \( \ell_3 \) gives rise to the edge \( \ell_1 \xrightarrow{M} \ell_3 \), without the need for the self-loop at \( \ell_1 \).

Thus even though the edge from \( \ell_1 \) to \( \ell_3 \) in Figure 2.14 may erroneously be added by applying the rule \( M \leftarrow s^p R^q \), the same edge can be added by rule \( M \leftarrow M M \) on path \( \ell_1 \rightarrow \ell_0 \rightarrow \ell_3 \), without using the self-loop \( s^p \).

The idea of using levels in the type derivation to avoid carrying the set of free type variables around appears standard in implementations of Hindley-Milner style type systems. Pessaux provides a nice description of the equivalence of level-based and non-level based inference of polymorphic types in his thesis ([Pes00], Chapter 8).
2.2.8 Demand-driven Algorithm

We now sketch a demand-driven CFL-closure algorithm using the ideas of [RHS95]. The grammar of Figure 2.13 is modified by deleting all epsilon productions and introducing an extra non-terminal $T$ for trigger edges. The trigger has the same function as start edges in [RHS95]. Figure 2.15 shows the resulting grammar for a backward demand algorithm (which requires a modification of the rules involving $L^0$). Given a label $\ell$, we ask for all labels that can potentially flow to $\ell$. We seed the algorithm with a trigger-edge $\ell \xrightarrow{T} \ell$ and an edge $\ell \xrightarrow{N} \ell$. Furthermore, we use three extra rules in the CFL algorithm:

<table>
<thead>
<tr>
<th>Given</th>
<th>Add</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\ell_1 \xrightarrow{\lambda} \bullet \xrightarrow{T} \bullet$</td>
<td>$\ell_1 \xrightarrow{T} \ell_1$ (D1)</td>
</tr>
<tr>
<td>$\ell_1 \xrightarrow{z} \bullet \xrightarrow{T} \bullet$</td>
<td>$\ell_1 \xrightarrow{T} \ell_1$ (D3)</td>
</tr>
<tr>
<td>$\ell_1 \xrightarrow{N} \bullet$</td>
<td>$\ell_1 \xrightarrow{P} \ell_1$ (D3)</td>
</tr>
</tbody>
</table>

Rules D1 and D2 add a trigger edge across an instantiation or self-loop edge allowing the closure to proceed. Rule D3 adds a $P$ edge to any node with an outgoing $N$ edge, allowing the derivation of $S$ paths. When no more new edges can be added to the graph closure, the query answer is the set of labels $\ell'$ connected to $\ell$ with an $S$-edge.

2.3 Soundness

Soundness of our flow relation $I; C \vdash_{\text{CFL}} \ell_1 \sim \ell_2$ is non-obvious and requires proof, which can be found in Appendix A. The presence of polymorphic recursion is a complicating factor, and it renders the proof non-trivial. The proof essentially consists in showing that, for every derivation in POLYFLOW$_{\text{CFL}}$, there exists a derivation in POLYFLOW$_{\text{copy}}$ such that our notion of flow is a safe approximation of the flow defined by the copy-based system (in fact, our notion of flow is equivalent, in terms of precision). The system POLYFLOW$_{\text{copy}}$ is similar to a standard copy-based system studied by Mossin for which soundness has been established [Mos96]).
A technical core in our proof consists in showing that the CFL-based flow relation can recover all substitutions on constraint systems used in the copy-based derivation. Since soundness has been established for copy-based systems by Mossin [Mos96], soundness of our notion of flow follows.

The soundness theorem states that for every derivation of \textsc{Polyflow}_{CFL}, there exists a derivation in \textsc{Polyflow}_{Copy} containing no more flow than the flow present in the \textsc{Polyflow}_{CFL} derivation. The proof is given in Appendix A.

**Theorem 2.2 (Soundness)** For every judgment
\[ t; I; C; A \vdash_{CFL} e : \sigma \]
derivable in \textsc{Polyflow}_{CFL} there exists a judgment
\[ C_G; I_G; C_G; A_G \vdash_{cp} e : \sigma \]
derivable in \textsc{Polyflow}_{Copy} such that, for all labels \( \ell \) and \( \ell' \) occurring in \( e \) one has
\[ I_G; C_G \vdash_{cp} \ell \leadsto \ell' \Rightarrow I; C \vdash_{CFL} \ell \leadsto \ell' \]

### 2.4 Discussion and Related Work

#### 2.4.1 Polymorphic Subtyping

Our work owes much to Mossin’s work on flow analysis based on structural polymorphic subtyping systems [Mos96]. In particular, we build on the soundness proof provided there for \textsc{Polyflow}_{Copy}, and use the same idea of introducing formal joins of labels to proof soundness of \textsc{Polyflow}_{CFL} with respect to \textsc{Polyflow}_{Copy}.

#### 2.4.2 Higher-Order Context-Sensitivity

Type-polyomorphic analyses are context-sensitive in the type abstraction graph of a program, not in the actual function call-graph as is standard in first-order precise inter-procedural
letrec P = λa.λb.
  if0 a then
    let b₁ = c.ℓ_c in
    let a₁ = (a - b₁)_{c+1} in
    let q = Pᵃ a₁ b₁ in
    let b₂ = q.2^{k+2} in
    q
  else p
  in

let main = λx. P^x 0^{o}

Figure 2.16: Example from [RHS95] (some edges omitted)

analyses (see discussion in Section 1.4). In the case where type abstraction and value abstraction coincide (first-order programs), type-polymorphic context-sensitivity coincides with the standard call-graph context-sensitivity. In the higher-order case, calls to lambda-bound functions are monomorphic within the body of the lambda. However, in many cases, context sensitivity is recovered by the fact that the higher-order function itself is polymorphic. Consider:

let f = λp.λx. ... p x ...
in
  let a = f g ...  let b = f h ...
...

Even though the call p x within the body of f is monomorphic, the applications f g and f h are fully polymorphic, since f's type is polymorphic. Thus there is no spurious flow between the parameters and results of g and h as might be expected.

38
2.4.3 Precise Interprocedural Dataflow Analysis

In [RHS95], Reps et. al. show how to solve a class of distributive interprocedural dataflow problems (called IFDS problems) precisely, i.e., restricted to valid call-return paths. Our POLYFLOW_{CFL} analysis is not directly comparable to the IFDS framework. They differ in at least the following points.

1. The IFDS problem is formulated for first-order programs only, whereas POLYFLOW_{CFL} deals directly with higher-order programs.

2. The IFDS problem does not deal with data structures (no data-flow facts are derived for data structure components), whereas POLYFLOW_{CFL} handles data structures directly.

3. On the other hand, the IFDS framework is flow-sensitive, i.e., there is an implicit store and data flow facts express information about the store at a particular program point. In contrast, our analysis is flow-insensitive, i.e., there is no concept of a store that is updated.

For POLYFLOW_{CFL} to deal with imperative languages, two avenues exist. Either POLYFLOW_{CFL} handles imperative features the way ML’s type system handles references, namely by treating them in a flow-insensitive way, i.e., all updates to a particular abstract location are accumulated. Alternatively, the program is transformed into an interprocedural SSA form. POLYFLOW_{CFL} applied to the SSA transformed program yields flow-sensitive information. Thus the problem in computing flow-sensitive information is not in the flow computation per se, but in obtaining a suitable interprocedural SSA representation of the program—a difficult task since it requires aliasing information in the presence of pointers and data structures.

Focusing more on the similarities, we can state that instead of deriving a subset of $|D|$ data-flow facts at each program point, POLYFLOW_{CFL} derives a single global set of $|D|^2$ data flow facts, namely the reachability relation between labels. Alternatively, we can view each label $\ell$ occurring in the program expression $e$ as a program point. Then the set of data flow facts derived for point $\ell$ is the set of reachable labels $L_i$ for each label $L_i$ appearing in the type labeled by $\ell$. However, the analogy does not express the fact that the paths in the exploded supergraph of [RHS95] carry only a single bit, namely true or false, whereas our paths carry sets of labels. There is thus a further distinction in what paths represent.

Figure 2.16 shows an SSA translation of the running example in [RHS95] into our language and the resulting type instantiation graph. Pairs are used to return the current values of the two SSA transformed variables $x$ and $y$. Instead of read statements for $x$ and $y$, we assume $x$ to be a parameter to main, and we introduce a constant $c$ for the read result assigned to $y$. Furthermore, we use 0 at label $\ell_0$ to represent the uninitialized value, and in order to model how uninitialized values propagate, we assume that the subtraction operation adds flow constraints from both arguments to the result. The question posed by Reps is whether $b$ can be uninitialized after the recursive call to $P$ at $j$. In our translation, the question is: Can $\ell_0$ flow to $\ell_j$? In order to answer this question, $P$ has to be treated polymorphically in the second argument $b$, saying that $P$ returns its second argument or $c$. One path connecting $\ell_0$ with $\ell_j$ is

$$
\ell_0 \xrightarrow{\ell_j} \ell_b \xrightarrow{\ell_1 \bullet \ell_j} \ell_j
$$

39
but it is invalid, since \((i \text{ does not match } j)\). The other three paths are not accepted by the grammar either. Thus, there is no flow from \(\ell_0\) to \(\ell_b\), and we can conclude that \(b_2\) is initialized on all paths.

### 2.4.4 Summarization in Context-Sensitive Analyses

Many context-sensitive flow analyses based on function summaries (e.g., [CRL99, LH99, FFA99]) are presented as two phase computations. In phase 1, information is propagated from the callees (where it originates) to the callers. In POLYFLOW\textsubscript{CFL}, this information is captured as positive flow \(P\). In phase 2, information is propagated from callers back to callees. The information in the second step represents summary information for a callee from all contexts. In our formulation, this final information is captured as \(PN\) or \(S\)-flow.

Our results show that this phase distinction is present on each individual flow path. As a result, it is not necessary to compute two global phases. Instead, we can use demand-driven algorithms. In the global 2-phase approach, it is easy to use demand-driven algorithms for phase 2 only. However, in that case, the information gathered in phase 1 already has size \(O(n^2)\) in the worst case.

Mossin’s formulation of POLYFLOW is also a 2-phase algorithm ([Mos96, pages 90,91]). His formulation relies crucially on the fact that the types include a set of polymorphic flow constraints that are \textit{copied} on instantiation, including constraints of the form \(L \leq \ell\), where \(L\) is a set of label constants. Type inference constitutes phase 1, after which flow information is present for the top-level function in the form of constraints involving constant labels. In phase 2, this information is propagated back into polymorphic functions using the reverse of substitutions performed during type inference. This step corresponds directly to our use of \(n\)-edges. Again, this approach does not yield a demand-driven algorithm and the information present after phase 1 has worst case size \(O(n^2)\).

We are not aware of any context-sensitive analysis computing summary information for every point in the program that deals directly with higher-order programs while generating only \(O(n)\) instances.

Context-sensitive analyses based on reanalyzing function bodies in different contexts do not have the problem of computing summary information without destroying context-sensitivity, since they enumerate each distinct context and can accumulate information at the same time. However, these analyses are typically exponential and use an arbitrary limit on the number of contexts being distinguished (e.g., [JW95, NN97]).

### 2.4.5 Constraint Simplification

Program analyses based on polymorphic subtyping depend crucially on constraint simplification in order to be at all practical [FM88, FA96, Pot96, Mos96, FF97, Pot98]. In the particular case of Mossin’s algorithm for inferring derivations of POLYFLOW\textsubscript{CFL}, the constraint set \(C\) is simplified before forming the quantified type \(\forall \ell \cdot C \Rightarrow \sigma\). Without this simplification, the constraint sets can grow exponentially in \(n\) due to the copying. In the case of polymorphic recursion, simplification is absolutely crucial to obtain a fixpoint for the polymorphic constrained type. The simplification consists in retaining only those constraints in \(C\) that involve free labels, or labels occurring in \(\sigma\), while retaining all flow implied by \(C\). This reduces to removing intermediate labels, while adding enough transitive constraints.

Since POLYFLOW\textsubscript{CFL} does not copy constraint systems, it automatically avoids the exponential blowup. The correspondence is yet closer, since 1) instantiation constraints only
relate labels appearing in \( \sigma \) and its instance, and 2) the transitive rule [Trans] together with rule [Match] in the flow relation \( \mathcal{F}_{\text{CFL}} \) derive the simplified constraints directly that are copied in \( \text{POLYFLOW}_{\text{CFL}} \). Interestingly, the fixpoint computation for polymorphic recursive functions is avoided altogether. It appears indirectly in the CFL algorithm where the fixpoint is computed for the set of non-terminal edges that can be added to the graph.

CFL-based approaches do not obviate all constraint simplification. For example, S-simplification [FM88] may be useful in obtaining smaller polymorphic types (i.e., fewer distinct labels) and thus fewer instantiation constraints. Furthermore, the computation of the CFL-closure may benefit from techniques such as online cycle elimination on M-edges [FFSA98].

### 2.4.6 Other Related Work

Henglein’s work on semi-unification and polymorphic recursion [Hen93] provides the logical foundation for our work. Dussart, Henglein and Mossin [DHM95a] present an algorithm for binding time analysis using polymorphic recursion with subtyping constraints, which uses constraint copying.

The unpublished work [DHM95b] formulates [DHM95a] purely in terms of semi-unification. However, it does not make the connection to CFL-reachability and does not address the computation of flow information. We believe our formulation of \( \text{POLYFLOW}_{\text{CFL}} \) is the first analysis combining instantiation (semi-unification) constraints and directional flow edges. In addition, we extend instantiation constraints with polarities and provide them with a flow interpretation.

Our work is also related to the closure analysis for ML by Heintze and McAllester [HM97]. The system they study corresponds to a type-based flow analysis with subtyping but without polymorphism. Such a system has also been studied by Mossin [Mos96]. The lack of polymorphism and context-sensitivity results in simple graph reachability queries answerable in linear time. Our approach degrades gracefully into a non-context sensitive setting. Given \( G_{\text{CFL}} \), individual queries can be answered in linear time by ignoring the matching problem and treating all edges as unlabeled flow edges.
Chapter 3

Flow and Type-Structure Polymorphism

In this chapter we extend the flow analysis of Chapter 2 with polymorphism over type structure. We extend the definitions from Figure 1.3 as shown in Figure 3.1. The language of types now contains type variables \( \alpha \). Type annotation on let- and letrec-bindings take the form \( \forall \alpha . \tau \). Labeled types \( \sigma \) are extended to type variables, where type variables are annotated with a special kind of label \( h \) called a \textit{hole-label}. Label sequences thus contain both ordinary labels \( \ell \) and hole-labels \( h^\alpha \), where the hole-labels are tagged with the type variable that they annotate in the type.

The well-labeling logic of Figure 1.4 is extended with the rule

\[
\Gamma \vdash \alpha \langle h^\alpha \rangle : \alpha^h
\]

and similarly, the subtype relation from Figure 1.6 is extended with the following rule.

\[
C \vdash h_1 \leq h_2 \quad \Rightarrow \quad \Gamma \vdash \alpha^h \leq \alpha^{h_2}[\text{Var}]
\]

In the same vein, we extend the instantiation relation of Figure 1.7 to type variables:

\[
I \vdash h \leq^i L \quad \Rightarrow \quad I \vdash \alpha^h \leq^i \tau(L,s)[\text{Var}]
\]

(Expressions)

\[
e \; ::= \; x \mid n \mid (e_1, e_2) \mid \lambda x. \tau . e \mid e_1 \; e_2 \mid
\]

let \( f : \forall \alpha . \tau = e_1 \) in \( e_2 \mid f^i \mid
\]

letrec \( f : \forall \alpha . \tau = e_1 \) in \( e_2 \mid
\]

if \( e_0 \) then \( e_1 \) else \( e_2 \mid .i
\]

(Types)

\[
\tau \; ::= \; \text{int} \mid \tau \rightarrow \tau \mid \tau \times \tau \mid \alpha
\]

(Labelled Types)

\[
\tau(s), \sigma \; ::= \; \text{int}^\ell \mid \sigma \rightarrow^\ell \sigma \mid \sigma \times^\ell \sigma \mid \alpha^h
\]

(Labels)

\[
L \; ::= \; \ell \mid h
\]

(Label Sequence)

\[
s, t \; ::= \; \ell \mid h^\alpha \mid s \; s
\]

Figure 3.1: Definitions
Note that we now have instantiation constraints on type variables of the form $\alpha \leq h^\tau$. Substitutions $\varphi$ now have three components: $\varphi|_\alpha$ maps type variables $\alpha$ to unlabeled types $\tau$, $\varphi|_{\ell^s}$ maps ordinary labels $\ell$ to ordinary labels $\ell^s$, and $\varphi|_h$ maps hole labels to label sequences $s$. The domain of a substitution $\varphi$ is the set of type variables and labels on which $\varphi$ is not the identity. We say that a substitution $\varphi$ is well-formed for a particular type $\sigma = \tau(s)$, if the result $\sigma'$ of the substitution is well-labeled, i.e. $\vdash \varphi(\tau)(\varphi(s)) : \sigma'$. In other words, for all $\alpha \in \tau$ and for all $h^\alpha \in s$, we have $\vdash \varphi(\alpha)(\varphi(h^\alpha)) : \text{wl}$, i.e., the structure substitution of a type variable $\alpha$ and the sequence substitutions of all hole labels tagged with $\alpha$ yield well-labeled types.

The notation $\tau(s)$ enables us to capture the substitution of multiple occurrences of type variable $\alpha$ at $h^\alpha_1, h^\alpha_2, \ldots$ with a single new structure $\tau$, while labeling the occurrences of $\tau$ at $h^\alpha_1, h^\alpha_2, \ldots$ differently. For example, the substitution $\text{int} \times \text{int}/\alpha[\ell_1 \ell_2 \ell_3/h_1^\alpha][\ell_4 \ell_5 \ell_6/h_2^\alpha]$ applied to type $\alpha^{\ell_1} \rightarrow \ell_0 \alpha^{\ell_5}$ yields

$$\text{int}^{\ell_2} \times \text{int}^{\ell_3} \rightarrow \ell_0 \text{int}^{\ell_5} \times \text{int}^{\ell_6}$$

3.1 CFL-Based Approach

We skip the copw-based approach and directly show how to extend the CFL-based approach of POLYFLOW_{CFL} to POLYTYPE_{CFL}.

Polymorphic types

Polymorphic types are now written $\forall \alpha \ell_1 h_1 \tau(s)$, with quantified variables $\alpha_1$ and quantified labels $\ell_1, h_1$. A quantified type is said to be well-labeled (written $\vdash \forall \alpha \ell_1 h_1 \tau(s) \text{wl}$), if for all $\alpha_1$, all hole-variables $h^\alpha_2 \in s$ are among the quantified labels $h_1$. In other words, hole-labels annotating occurrences of quantified variables $\alpha$ must themselves be quantified. We use the notation

A type $\tau'(s')$ is an instance of a polymorphic type $\forall \alpha \ell_1 h_1 \tau(s)$, written $\forall \alpha \ell_1 h_1 \tau(s) \leq \tau'(s')$, if there exists a substitution $\varphi$ over the quantified variables, such that $\varphi(\tau) = \tau'$, $\varphi(s) = s'$, and $\vdash \tau'(s') \text{wl}$. The set of free hole labels of types and environments are written $fh(\sigma)$ and $fh(A)$.

3.1.1 Type Rules

Figure 3.2 shows the rules for the type polymorphic flow system. The base rules are identical to the rules for POLYFLOW_{CFL} in Figure 2.12. The polymorphic rules now quantify over type variables and instantiate them to types at instantiations. Note that we use the annotations of the underlying structural polymorphic types at [Let] and [LetRec]. Finding a polymorphic typing derivation for the underlying structural types can be done using algorithm $W$, provided that polymorphic types in the letrec case are given through some other means (programmer, or semi-decidable procedure [Hen93]).

For completeness, the type rules are presented with the level optimization and summary self-loops discussed in Section 2.2.7. In the remainder of this chapter however, we will elide summary self-loops and level annotations on instantiation constraints, since they are orthogonal to the issues discussed here.
\[ I; C; A \vdash^p e : \sigma \]

**Base Rules**

\[ I; C; A, x : \sigma \vdash^p t : \sigma \quad [\text{Id}] \]

\[ I; C; A \vdash^p \text{int}^\ell : \text{int}^\ell\quad [\text{Int}] \]

\[ I; C; A \vdash^p e_1 : \sigma_1 \quad I; C; A \vdash^p e_2 : \sigma_2 \]

\[ I; C; A \vdash^p e_1 : \sigma_2 \rightarrow^\ell \sigma_1 \quad I; C; A \vdash^p e_2 : \sigma_2 \quad [\text{App}] \]

\[ \vdash \tau(s) : \sigma \quad t s; I; C; A, x : \sigma \vdash^p e : \sigma' \]

\[ I; C; A \vdash^p \lambda x : \tau. e : \sigma \rightarrow^\ell \sigma' \quad [\text{Lam}] \]

\[ I; C; A \vdash^p e_1 : \sigma_1 \quad I; C; A \vdash^p e_2 : \sigma_2 \]

\[ I; C; A \vdash^p (e_1, e_2) : \sigma_1 \times^\ell \sigma_2 \quad [\text{Pair}] \]

\[ I; C; A \vdash^p e : \sigma_1 \times^\ell \sigma_2 \quad [\text{Proj } i = 1, 2] \]

\[ I; C; A \vdash \text{if } e_0 \text{ then } e_1 \text{ else } e_2 : \sigma \quad [\text{Cond}] \]

\[ I; C; A \vdash^p e : \sigma \quad C \vdash \sigma \leq^\ell \sigma' \quad [\text{Sub}] \]

\[ \vdash \tau(\ell s) : \sigma \quad C \vdash \ell = \ell' \quad [\text{Label}] \]

**Polymorphic Rules**

\[ I; C', A \vdash^p e_1 : \sigma_1 \quad I; C, A, f : (\forall \alpha \check{\sigma}, \sigma_1, k) \vdash^p e_2 : \sigma_2 \]

\[ C \vdash C' \quad \ell = \text{gen}_e(A, \sigma_1) \quad \ell = \text{gen}_h(A, \sigma_1) \]

\[ \vdash \tau(s) : \sigma_1 \quad +\forall \alpha \check{\sigma}, \sigma_1 \text{\ inner} \quad [\text{Let}] \]

\[ I; C; A \vdash^p \text{let } f : \forall \alpha. \tau = e_1 \text{ in } e_2 : \sigma_2 \quad [\text{Rec}] \]

\[ I; C, A, f : (\forall \alpha \check{\sigma}, \sigma_1, k) \vdash^p e_1 : \sigma_1 \quad I; C, A, f : (\forall \alpha \check{\sigma}, \sigma_1, k) \vdash^p e_2 : \sigma_2 \]

\[ C \vdash C' \quad \ell = \text{gen}_e(A, \sigma_1) \quad \ell = \text{gen}_h(A, \sigma_1) \]

\[ \vdash \tau(s) : \sigma_1 \quad +\forall \alpha \check{\sigma}, \sigma_1 \text{\ inner} \quad [\text{Rec}] \]

\[ I; C; A \vdash^p \text{letrec } f : \forall \alpha. \tau = e_1 \text{ in } e_2 : \sigma_2 \quad [\text{Rec}] \]

\[ \vdash \sigma \leq^p \varphi \quad \text{dom } \varphi = \check{\alpha}, \check{\ell}, \check{h} \quad [\text{Inst}] \]

\[ \text{gen}_e(A, \sigma) = f l(\sigma) \setminus f l(A) \]

\[ \text{gen}_h(A, \sigma) = f h(\sigma) \setminus f h(A) \]

**Figure 3.2: POLYTYPE_{CFL}**
let \( \text{id} = \lambda x: \alpha. \ x \) in
let \( f = \lambda y: \beta. \ \text{id}^i (a^i) \cdot \text{id}^k (y^i)^{\ell^p} \) in
let \( z = (f^j \ b^i).2^i \);

...
\[ S \leftarrow PN \]
\[ P \leftarrow MP \]
\[ \quad \mid pP \]
\[ \quad \mid \epsilon \]
\[ N \leftarrow MN \]
\[ \quad \mid nN \]
\[ \quad \mid \epsilon \]
\[ M \leftarrow (iM)_i \]
\[ \quad \mid \left[ \begin{array}{c} c_i \\ M \end{array} \right] e_i \quad \text{covariant in } c \]
\[ \quad \mid \left[ \begin{array}{c} o \\ M \end{array} \right] e_i \quad \text{contravariant in } c \]
\[ \quad \mid d \]
\[ \quad \mid \epsilon \]

Figure 3.4: Extended grammar for constructor matching

ith child of a type constructors \( c \) with the edge from \( c \) to its ith child whenever the ith child of a constructor has the same variance as the constructor itself. The second production handles the contravariant case. The non-terminal \( \overline{M} \) stands for reversed \( M \) paths. We choose to express these paths externally from the grammar by an extra rule

\[
\text{if } \ell_1 \xrightarrow{M} \ell_2 \text{ is an edge in } G_c, \text{ then } \ell_2 \overline{M} \ell_1 \text{ is also an edge in } G_c. \quad \text{(RevM)}
\]

It is possible to express reversed \( M \)-paths directly in the grammar, but we have to introduce reverse edges for all terminals involved in \( M \) productions and write corresponding productions for \( \overline{M} \) as follows:

\[
\overline{M} \leftarrow \overline{M} \overline{M}_i
\]

In general, constructor and instantiation matching are independent and give rise to an interleaved CFL problem. Deciding whether there is a single path accepted by both CFL problems is undecidable [Rep00]. Fortunately, we can prove for our particular flow analysis that there always exists a flow path with properly nested parentheses of both kinds, resulting in a single CFL problem. Consider for example the alternative path in Figure 3.3 formed by

\[
\ell_b \xrightarrow{u} \ell_y \xrightarrow{a} \ell_z \xrightarrow{h_1 \rightarrow h_2 \rightarrow \cdots \rightarrow h_3} \ell_1 \xrightarrow{x_2} \ell_2 \xrightarrow{h_1 \rightarrow h_2 \rightarrow \cdots \rightarrow h_3} \ell_3 \xrightarrow{x_2} \ell_z
\]

This path differs from our earlier path in the last two edges traversed. Instead of traversing the edge \( \ell_1 \xrightarrow{x_2} \ell_2 \) followed by \( \ell_2 \xrightarrow{y} z \), we take \( \ell_1 \xrightarrow{y} \ell_3 \) followed by \( \ell_3 \xrightarrow{x_2} z \). The second path matches what happens operationally, namely that we return the pair from function \( f \) along the edge \( \ell_1 \xrightarrow{y} \ell_3 \) and then select the second component of the pair. However, on this path, the constructor and instantiation parentheses are not properly nested, since \( [x_2 \text{ does not match up with } ]_j \).

If we interpret the first flow path operationally, we see that we select the second component of the pair within function \( f \) prior to returning it in order to properly nest \( [x_2 \text{ and } ]_j \). The reason we can always do this is that we work with a structural
subtyping system, where subsumption steps $\sigma_1 \leq \sigma_2$ always require that the type structure of $\sigma_1$ and $\sigma_2$ are the same. This results in the following property:

**Lemma 3.1** If there is matched flow $C \vdash \ell_1 \sim_m \ell_2$ from label $\ell_1$ to $\ell_2$ and $\ell_1$ annotates $\tau_1$ and $\ell_2$ annotates $\tau_2$, then $\tau_1 = \tau_2$.

Although we do not have a direct soundness proof for the POLYTYPE system, we believe that at least for the case where type polymorphism is restricted to non-recursive ML-polymorphism, the proof of soundness can be reduced to the proof of POLYFLOW by examining the monomorphic type expansion. A proof of the recursive polymorphic case is left as future work.

### 3.1.3 Practical Aspects of Constraint Generation

Given the family of extra grammar productions $M \leftarrow [\sigma_i \ M \ ]_c$, and $M \leftarrow [\sigma_i \ Mt_i \ ]_c$, it is no longer necessary for the subtype relation $C \vdash \sigma \leq \sigma'$ to relate labels at all levels of $\sigma$ and $\sigma'$, since these relations can be derived indirectly as CFL-reachability via the extra productions. This is an important aspect in practice, for it allows us to derive yet fewer constraints when generating the initial type instantiation graph. Taking this idea to its logical extreme suggests that we also delay the downward closure of instantiation constraints. Instead of relating every label in $\sigma \leq^i \sigma'$ via an instantiation constraint, we only relate the labels of the top-level constructor in $\sigma$ and $\sigma'$. The remaining relations can be recovered through this one and the extra grammar productions spelled out below (and similarly for $p$ and $n$ edges):

- $\ell_i \leftarrow [\ell_i \ | \ell_i]_{c_i}$ $c_i$ covariant in $c$
- $\ell_i \leftarrow [\ell_i \ | \ell_i]_{c_i}$ $c_i$ contravariant in $c$
- $\ell_i \leftarrow [\ell_i \ | \ell_i]_{c_i}$ $c_i$ covariant in $c$
- $\ell_i \leftarrow [\ell_i \ | \ell_i]_{c_i}$ $c_i$ contravariant in $c$

Figure 3.5 illustrates the application of the first two rules to a function type constructor. The original instantiation edge $\ell_0 \to \ell'_0$ gives rise to the two edges $\ell_2 \to \ell_3$ and $\ell'_1 \to \ell_1$.

Recall that given a program of size $m$, the number $n$ of distinct labels and constraints generated in the type-monomorphic case is exponential in $m$ in the worst case. The reason for the exponential difference is that the unlabeled type structure can be represented as a DAG with size $O(m)$, whereas the labeled type structure is a tree of the same shape, because shared type structures are labeled independently.

If we delay the downward closure of subtyping and instantiation steps during constraint generation as outlined above and generate labels on types only on demand, the number of distinct labels, flow, and instantiation edges is bounded by $O(m)$.

In the type-polymorphic case, the size of the unlabeled type graph can itself be exponential in $m$, and the labeled type graph is then doubly-exponential in $m$. However, experience with ML typing has shown that in practice this situation does not arise. In any case, using the delayed strategy for flow and instantiation constraints, we still generate only $O(m)$ edges and labels. Of course, any query of the system may potentially involve closing the graph, and thus generating the exponential or doubly exponential number of labels and edges in the worst case. In practice we expect the demand-driven queries to explore only small parts of the labeled structure.
We haven’t sketched how the demand-driven strategy decides what labels to quantify over. The trick is to choose a derivation where we can quantify over every label in the quantified type by using the same trick used in the proof of the soundness theorem, namely that we use a subsumption step in the let and letrec rules to freshen the labels of $\sigma_1$. This guarantees that any label in $\sigma_1$ can be instantiated to a fresh label at all instances.

Such a demand-driven scheme can be implemented by using two data structures with the following signatures:

\[
T : L \rightarrow \tau \\
M : L \rightarrow N \rightarrow L
\]

Structure $T$ is a map from labels to types. For this map to exist, we require that each label annotate a unique type. The second map $M$ is used to traverse the labeled type structure. Given a label $\ell_1$ and a child index $k$, $M \ell k$ produces the label annotating the $k$th child position of the labeled type annotated by $\ell_1$. Initially, the two maps are empty. As constraint derivation proceeds, the mappings $T$ and $M$ are extended necessary. The original size of $T$ and $M$ will be $O(m)$. During CFL-closure for a particular query, the maps $M$ and $T$ are consulted and further extended as necessary.

## 3.2 POLYEQ: An Efficient Special Case

Context-sensitivity based on Hindley-Milner style polymorphic type inference is in wide spread use due to its good practical running time. The low cost of such inference based analyses stems from the use of unification to model intra-procedural dependencies of values. Inter-procedural dependencies of values is captured by instantiations of polymorphic function types, but this information is generally ignored.

Here we view Hindley-Milner style polymorphism as a special case of POLYTYPE with a number of restrictions that allow for a more efficient flow algorithm. The resulting system is called POLYEQ and has the following characteristics:

- The type system does not admit the subsumption rule [Sub], thus there are no flow constraints $C$.  


• Each polymorphic type variable $\alpha$ has a single unique hole label $h^\alpha$ used at every occurrence of $\alpha$.

• Individual all sources—one sink flow queries can be answered in time $O(n)$ in the size of the type instantiation graph.

In the absence of subtyping, the flow of values within an instantiation context is entirely modeled by equivalence classes. The flow of values between different instantiation contexts is characterized by instantiation edges, which are directed.

### 3.2.1 Type System

As described above, the type system of POLYEQ differs from POLYPE only in the omission of the [Sub] rule, along with the restriction that if two hole variables $h^\alpha_1$ and $h^\alpha_2$ are associated with the same type variable $\alpha$, then $h^\alpha_1 = h^\alpha_2$. We thus do not restate the rules here.

### 3.2.2 Flow Computation

The computation of flow queries on $G_{CFL}$ degenerates from a CFL-reachability problem to an RE-reachability problem (regular expression). To obtain the form of the RE-reachability problem, consider the grammar in Figure 3.4. We reproduce the productions for non-terminal $M$ representing matched flow below:

$$ M \leftarrow (iM)_i \\
| (\alpha, M)_{\alpha_i} c_i \text{ covariant in } c \\
| (\alpha, \overline{M})_{\alpha_i} c_i \text{ contravariant in } c \\
| M \quad M \\
| d \\
| \epsilon $$

Since we have no directed edges, the production $M \leftarrow d$ can be omitted. We now show by induction on the grammar reductions that the only $M$-edges produced by this grammar for a graph $G_{CFL}$ are self-loops. The base case for the epsilon production is trivial. Next, consider the production $M \leftarrow [\alpha_i, M]_{\alpha_i}$. By induction, the $M$ in the right-hand side represents a self-loop on a label $\ell$. Thus the edges labeled by $[\alpha_i]$ and $[\alpha_i]$ arise from the same constructor edge rooted at $\ell$ and form a loop. The $M$-edge produced is a self-loop on the other end-point of this loop, namely the $i$th child of $\ell$. The case for $M \leftarrow [\alpha_i, \overline{M}]_{\alpha_i}$ is analogous. The transitivity rule $M \leftarrow M M$ is trivial. Finally, consider $M \leftarrow (iM)_i$. Recall that this rule encodes the following situation of instantiation constraints

$$ \ell_1 \sim_{\mathbb{M}} \ell_2 \\
\ell_i \sim^+ \ell_i \\
\ell_1 \cdots \ell_2 $$

where we have $\ell_1 \sim_{\mathbb{M}} \ell_2$. By induction, we have $\ell_1 = \ell_2$. Since instantiation edges represent substitutions, we must have $\ell_1 = \ell_2$. Thus the added $M$-edge on $\ell_1$ forms a self-loop.
<table>
<thead>
<tr>
<th>Test program</th>
<th>Code lines</th>
<th>AST nodes</th>
<th>Ave. deref size</th>
<th>Analysis time (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mono</td>
<td>Poly</td>
<td>Mono</td>
<td>Poly</td>
</tr>
<tr>
<td>compress</td>
<td>1,904</td>
<td>2,234</td>
<td>7</td>
<td>0.5</td>
</tr>
<tr>
<td>li</td>
<td>7,602</td>
<td>23,379</td>
<td>282.1</td>
<td>1.8</td>
</tr>
<tr>
<td>m88ksim</td>
<td>19,412</td>
<td>65,967</td>
<td>107.3</td>
<td>4.6</td>
</tr>
<tr>
<td>jpeg</td>
<td>31,215</td>
<td>79,486</td>
<td>37.9</td>
<td>7.0</td>
</tr>
<tr>
<td>go</td>
<td>29,919</td>
<td>109,134</td>
<td>51.3</td>
<td>4.7</td>
</tr>
<tr>
<td>perl</td>
<td>26,871</td>
<td>116,490</td>
<td>51.1</td>
<td>7.6</td>
</tr>
<tr>
<td>vortex</td>
<td>67,211</td>
<td>200,107</td>
<td>1661.3</td>
<td>17.3</td>
</tr>
<tr>
<td>gcc</td>
<td>205,406</td>
<td>604,100</td>
<td>429.5</td>
<td>42.3</td>
</tr>
</tbody>
</table>

Table 3.1: Raw measurement data: Lines of code and AST node count. Average sizes of points-to sets at static dereference points, and running time in seconds.

Given that $M$ only derives empty strings, we can simplify the grammar considerably to

$$
S \leftarrow PN \\
P \leftarrow pP \\
N \leftarrow nN \\
\mid \epsilon
$$

This grammar only accepts sequences formed by the regular expression $p^*n^*$. We call such paths $PN$-paths. $PN$-path reachability can be computed in $O(n)$ time for a graph of size $n$ ($n$ nodes, $O(n)$ edges) for all sources—one sink, or all sinks—one source queries. Computing all queries takes $O(n^2)$ time.

### 3.2.3 Practical Applications

Given the restrictions on POLYEQ, derivations become equivalent to their underlying Hindley-Milner or Milner-Mycroft type derivation. There is a one-to-one mapping between type nodes and labels. Thus, labels need no longer be explicitly represented. All that is needed are the instantiation constraints corresponding to the substitutions on the type structure at instantiation sites. This opens up the possibility of retrofitting the above flow analysis onto existing type analyses based on Hindley-Milner style polymorphism.

As a practical application, we use POLYEQ to compute points-to information and function pointer information for the C programming language. To show that our techniques scale to large programs we present numbers for an implementation of the described type inference and flow algorithms. We show the precision improvements gained over a monomorphic version in the context of points-to analysis. The monomorphic version corresponds to Steensgaard’s analysis \cite{Steensgaard1996}.

We analyze a range of C programs from the SPEC benchmark suite. The raw numbers are given in Table 3.1. All experiments were run on a Dell Precision 610 with 512MB of memory. To measure the precision of the points-to analysis, we count points-to set sizes at static pointer dereference points only (direct accesses to arrays are not counted as dereferences).

\footnote{Library function stubs are treated polymorphically, no conditional unification is used.}
Figure 3.6: Reductions of points-to sizes

Figure 3.7: Running times

Figure 3.8: Space overhead

Figure 3.6 shows the reductions in the average points-to set size obtained through polymorphism. The most dramatic reduction is obtained for Vortex, where the average points-to set size drops from 1661 to 62. Even for GCC, we get almost a factor of 5 reduction in the average points-to set size.

Figure 3.7 gives the running time of the monomorphic and the polymorphic analyses. We give the time per abstract syntax tree node to show the scaling behavior. The running time is broken down into monomorphic running time, time for computing the polymorphic tree instantiation graph (in excess of the monomorphic time), and the time to compute the flow result. The numbers show that the polymorphic type instantiation graph can be computed with little overhead over a monomorphic analysis. The time to compute the flow information however is a substantial fraction of the analysis time. Fortunately, the absolute times are small (< 3mins for gcc). The flow computation is currently implemented as a forward-only flow, where each symbol is propagated along all PN-paths. We believe this naive implementation can be improved substantially.

Finally, Figure 3.8 shows the space consumption of the polymorphic analysis as a factor of the space consumption of the monomorphic analysis. The space is broken down into type nodes and instantiation edges. The space overhead of polymorphism is substantial and currently the main inhibitor to scaling the analysis to very large programs. We are able to construct the final type instantiation graph for MS Word (2.1MLOC) within 512MB of memory, but exceed memory during the flow computation. Finding ways to further reduce the memory consumption is part of future work.
let pair = λy: int. (a^{\ell_y}, y^{\ell_y})^{\ell_y} in
let z = (pair^{\ell_z} b^{\ell_z}), 2^{\ell_z}
...

Figure 3.9: Non-structural subtyping example

3.3 Extensions

3.3.1 Non-Structural Subtyping

So far, we have assumed that all subtyping steps are structural. Non-structural subtyping allows subtyping judgments of the form $\sigma_1 \leq \sigma_2$, where $\sigma_1$ and $\sigma_2$ do not necessarily have the same structure. An example of such a system is the type system based on set constraints presented in [AW92]. Non-structural subtyping can lead to the problem that although there is a path between two labels, every path contains interleaved matchings of instantiations and constructor labels.

Consider the non-structural case depicted in Figure 3.9 for a simpler example than the one in Figure 3.3. Non-structural subtyping can give the type $\forall \ell_y \beta h_2. \text{int}^{\ell_y} \rightarrow \beta^{y_2}$ to pair, along with a non-structural subtyping side constraint $\text{int}^{\ell_y} \times \text{int}^{\ell_y} \leq \beta^{y_2}$. At instance $i$, $\beta^{y_2}$ instantiates to the pair $\text{int}^{\ell_z} \times \text{int}^{\ell_z}$. In order to find flow from $b$ to $z$, we are forced to take the path with interleaved parentheses

$$\ell_y \xrightarrow{\ell_y} \ell_p \xrightarrow{x_2} \ell_y \xrightarrow{x_2} \ell_h \xrightarrow{x_2} \ell_1 \xrightarrow{x_2} \ell_z$$

As the example shows, non-structural subtyping cannot be allowed arbitrarily, but it is still possible to mix structural and non-structural subtyping as long as we can guarantee that if there is an interleaved path from $\ell_1$ to $\ell_2$ with matched instantiation and constructor labels, then there also exists a path with perfectly nested matchings of instantiation and constructor labels. The condition for such nested matchings to exist is simple.

**Conjecture 3.2** in the presence of non-structural subtyping, flow paths with perfect nesting of instantiation edges and constructor edges exist as long as the type structure of any type annotated with a quantified label $\ell$ is fully expanded to the underlying Milner-Mycroft type structure.

This requirement is weaker than the one of Lemma 3.1. It guarantees only that if there is matched flow $\ell \sim_m \ell$ or $\ell \sim_m \ell'$, and $\ell$ is a quantified label (i.e., it appear to the left
of some instantiation constraint), then the type structure annotated by $\ell$ is as concrete as the type structure annotated by $\ell'$. We also make the implicit assumption that no label annotates two structurally distinct types.

Our formulation of the type rules [Let] and [Letrec] already require the quantified type $\sigma_1$ to be as expanded as the underlying Milner-Mycroft type $\tau$. It is thus already admissible in our type system to use non-structural subtyping steps in the subsumption rule [Sub].

One can relax the structural requirements on types even further by observing that nested matchings need only exist on paths connecting input with output labels. Consider again Figure 3.9 and the flow path from $\ell_a$ to $\ell_2$:

$$\ell_a \xrightarrow{\ell_1} \ell_p \rightarrow h_2^\beta \xrightarrow{\ell_0} \ell_1 \xrightarrow{\ell_1} \ell_2$$

Here we have an interleaving of p-edges and constructor edges. Such an interleaving is however not a problem, since $p$ (or $n$) edges do not enter into matched flow. It is possible to recognize such paths using the extra grammar rules below:

$$S \leftarrow P [_{\ell_1} \, S ]_{\ell_1} \, N$$
$$P \leftarrow [_{\ell_0} \, P ]_{\ell_0}$$
$$N \leftarrow [_{\ell_0} \, N ]_{\ell_1}$$

The minimal expansion of the type structure needed for the program in Figure 3.9 is therefore as follows:

Note that the pair type structure at $h_2^\beta$ only expands the right component. The left component does not need expansion. We capture these weaker requirements for existence of perfectly nested paths as follows.

**Conjecture 3.3 (Partial Labeling)** In the presence of non-structural subtyping, flow paths with perfect nesting of instantiation and constructor edges exist if for every input label $\ell_1$ and output label $\ell_2$ of the fully annotated polymorphic type $\forall \ell. \tau(s)$ such that there is matched flow from $\ell_1$ to $\ell_2$, the actual type used in the type instantiation graph must be expanded to contain at least $\ell_1$ and $\ell_2$.

The restriction to input and output labels makes clear that we only care about paths that may involve matched instantiation edges. Paths that involve $p$ and $n$ edges are handled by the extra grammar rules shown above. The view put forth here is that non-structural subtyping is a way of dealing with partially labeled types.

The advantage of using some non-structural subtyping steps in a derivation lies in that it may substantially reduce the size of the resulting type instantiation graph and the number of flow and instantiation edges.
3.3.2 Recursive Types

Recursive types describe regular infinite tree structures. Such structures have finite representations, for example as \(\mu\)-types of the form \(\mu \alpha. \tau\), where \(\alpha\) may occur in \(\tau\). Such a \(\mu\)-type encodes the equation \(\alpha = \tau\). Extra fold/unfold type rules describe the equivalence of infinite types with distinct representations. The standard rule for unfolding recursive types in non-annotated type systems is simply:

\[
\frac{A \vdash e : \mu \alpha. \tau}{A \vdash e : \tau[\mu \alpha. \tau/\alpha]}
\]

Viewed in the world of labeled types, recursive types give rise to infinite labeled types, where each node in the infinite tree may be labeled by a distinct label. Recursive types are only of interest in the presence of recursive functions that manipulate such infinite structures. Without recursive functions, any program can only inspect a finite prefix of these infinite trees, and therefore an explicit labeling is possible. This view is consistent with the view put forth in the previous section on non-structural subtyping. Even though the underlying type structure is fixed and possibly infinite, we deal with only partially labeled types.

In the presence of recursive functions, programs can build and inspect recursive data structures up to unbounded depth. In general, the partial labeling requirement put forth in Conjecture 3.3 leads to infinitely labeled types. For example, let \(\text{list} = \mu \alpha. \text{int} \times \alpha\) be a type for integer lists and let \(\text{nil}\) be the empty list value. The program

\[
\text{letrec double} = \lambda l. \\
\quad \text{if } \text{isNil} l \text{ then nil} \text{ else} \\
\quad \quad (l.1, (l.1, \text{double} \ l.2))
\]

takes an input list \(l\) and produces an output list where elements \(2i\) and \(2i+1\) are equal to element \(i\) of the input list. If we view the input type and the output type as infinite labeled trees, we see that there is matched flow from the input type to the output type at arbitrary depth.

As a result, it is not possible to expand the type structure only finitely and still obtain a perfectly nested matching problem. There seem to be two alternatives. First, expand the type structure as little as desired and deal directly with the interleaved matching problem. As stated earlier, the general interleaved matching problem is undecidable [Rep00]. The second alternative is to use a regular labeling of the infinite tree structure, where labels reused infinitely many times. Using the same label to annotate multiple type nodes at different depths of a recursive type introduces spurious flow between these type nodes. In practice, this approach is the most natural one to use. The underlying type structure already provides regular finite representations for the underlying infinite types in form of \(\mu\)-types. We can translate these representations into regular labeled \(\mu\)-types as follows. Given \(\mu \alpha. \tau\), let the regular labeled type \(\mu \alpha. \tau(s)\) be such that there exists a unique hole label \(h^\alpha\) in \(s\) labeling all occurrences of the bound variable \(\alpha\). Any unfolding of the regular labeled type is obtained via the following rule:

\[
\frac{C, A \vdash e : \mu \alpha. \tau(s) \quad \varphi(h^\alpha) = s \quad \varphi(L) = L \quad L \neq h^\alpha}{C, A \vdash e : \tau[\mu \alpha. \tau/\alpha](\varphi(s))} \quad \text{(Unfold)}
\]

The substitution \(\varphi\) expands the hole variable \(h^\alpha\) into the label sequence \(s\). Since \(s\) contains \(h^\alpha\) itself, this expansion mirrors the type structure expansion \(\tau[\mu \alpha. \tau/\alpha]\).
When approximating infinite labeled types with regular labeled types, the number and placement of fold/unfold steps in the program can have an influence on the precision. Minimizing the number of occurrences of fold/unfold steps through other means (e.g., tagging optimization [Hen92]) might be beneficial.

3.4 Related Work

Mossin briefly touches upon the subject of flow analyses in the presence of polymorphic type structure [Mos96]. Expressed in our formulation, e considers only two distinct hole variables $h_{\lambda}^+$ and $h_{\lambda}^-$ for each polymorphic type variable $\alpha$, where the first is used to annotate positive occurrences of $\alpha$, and the second for negative occurrences of $\alpha$. Such an approach approximates the flow of values by assuming that all input occurrences of a type variable $\alpha$ flow to all output occurrences of $\alpha$. For a function such as

$$\text{twice} : \forall \alpha. (\alpha \to \alpha) \to \alpha \to \alpha$$

where $\text{twice} \ f \ a = f (f \ a)$, the treatment he proposes yields strictly less precise flow than our approach in POLYTYPE.

In [Rep00], Reps provides an elegant proof for the undecidability of flow queries in the presence of interleaved call-return and constructor matchings. His presentation reduces a variation of the post-correspondence problem to a flow query. His result provides an excellent foundation for understanding the flow problems arising through the interaction of polymorphic subtyping and recursive types.

The extension of CFL-based flow analysis to type polymorphism is also related to work by O’Callahan and Jackson [OJ97]. They use polymorphic type inference for software analysis of C programs. They define a symmetric compatibility relation between values based on the instantiations of type variables. Their system is closest to POLYEQ. Our directed flow relation of POLYEQ is a strengthening of their symmetric compatibility relation, providing strictly better information at no extra cost.
Chapter 4

Conclusions

We presented a novel approach to computing context-sensitive flow of values through procedures and data structures. Our results are founded on a novel analysis of polymorphic subtyping, as a combination of instantiation (or semi-unification) constraints and subtyping constraints, and it includes polymorphic recursion.

The main contributions of this article are:

- A novel algorithm for computing context-sensitive, directional flow information for higher order typed programs. Our algorithm improves the asymptotic complexity of a known algorithm based on subtyping from $O(n^3)$ to $O(n^3)$. For intra-procedural flow restricted to equivalence classes, our algorithm yields linear time inter-procedural flow queries.

- Our algorithm is demand driven. We prove that context sensitive flow can be computed by CFL (Context Free Language) reachability in polymorphic subtyping systems. This result leads to a characterization of individual, valid context sensitive flow paths, and it allows us to answer single flow queries on demand. The initial constraint graph is linear in the size of the program.

- We transfer results on precise interprocedural dataflow analysis based on CFL reachability [RHS95] to the setting of type based analysis, resulting in an algorithm which works directly on higher order programs with structured data.

- Our results open the door to new implementation techniques for flow analyses based on polymorphic subtyping systems. By obviating the need to multiply copies of subtyping constraints, our technique may circumvent one of the main scaling inhibitors for such systems.
Bibliography


Appendix A

Soundness Proof

We prove soundness of the system POLYFLOW_{CFL}. We will do so by proving the system sound relative to POLYFLOW_{copy}; since soundness has been established for a system sufficiently similar to POLYFLOW_{copy} in [Moe96], the soundness of POLYFLOW_{CFL} follows from the soundness results of [Moe96].

A.1 Overview of the proof

The flow relation for a program e determined by a typing \( t; I; C; A \vdash_{\text{CFL}} e : \sigma \) in POLYFLOW_{CFL} is defined by the CFL-reachability relation \( I; C \vdash_{\ell} \ell' \) on the labels appearing in e. As we have seen, a central motivation for this relation is to rule out spurious flow paths, thereby gaining precision in the flow analysis. From the point of view of soundness, however, the problem is to ascertain that the flow relation does not rule out too many paths, so that any flow that could be realized during any possible execution of the program is accounted for. Hence, to prove soundness, we need to show that CFL-reachability is a safe approximation (overestimates) the real flow of the program. We will do this by showing that the flow relation safely approximates the flow relation defined by the system POLYFLOW_{copy}, which is sufficiently close to the system studied by Mossin for which soundness has already been established [Moe96]. In fact, we believe that the copy-based systems and the CFL-based system are equivalent with respect to precision, but we shall only be concerned with soundness here.

Technically, the proof consists in showing that, for every derivation in POLYFLOW_{CFL}, there exists a derivation in POLYFLOW_{copy} such that the CFL-based flow relation defined by the derivation in POLYFLOW_{copy} is a safe approximation of the flow defined by the derivation in the copy-based system. A technical core in the proof is to show that our notion of CFL-based flow can recover all substitutions on constraint systems used in some copy-based derivation. Suppose that we have a POLYFLOW_{CFL} derivation of \( t; I; C; A \vdash_{\text{CFL}} e : \sigma \). We wish to show that there exists a derivation \( C_r; I'; C'; A' \vdash_{\text{copy}} e : \sigma' \) in POLYFLOW_{copy} such that, intuitively, the subtyping relation on the labels in e defined by the latter derivation is safely approximated by the flow relation derivable from I and C under CFL-reachability. The intuition why the CFL-based system works is then this, that all flow, which is explicit in a copy-based system, can be recovered from the flow constraints (C) and the instantiation
constraints (I). As mentioned earlier, the central case we must be able to handle is this:

\[ \ell_1 \xrightarrow{\ell_{2}} \ell_2 \]

\[ \ell_3 \xrightarrow{\ell_{4}} \ell_4 \]

where flow from \( \ell_3 \) to \( \ell_4 \) is recovered by matched flow under CFL-reachability. This intuition provides the proof idea that we can construct the desired typing derivation of \( C' \); \( I'; C'; A'; e : \sigma' \) in POLYFLOW\_copy from the given derivation of \( t; I; C; A \vdash_{\text{CFL}} e : \sigma \) by closing the set \( C \) under matched flow, using the constraints in \( I \) and \( C \). Let \( C' \) denote this flow closure of \( C \), i.e., we take \( C' \) to be the set of all flow constraints of the form \( \ell \leq \ell' \) where \( I; C \vdash_{\text{CFL}} \ell \rightarrow_{\text{m}} \ell' \) holds. Then we will show that \( C' \) contains all the constraint copies needed for a derivation in the copy-based system. The technical property needed here is that, whenever a substitution \( \varphi \) is used at an instantiation site, then one has

\[ C' \vdash \varphi(C') \]  

(A.1)

that is, \( C' \) is closed under substitutions. This property will render \( C' \) applicable in the rule [Inst] of POLYFLOW\_copy.

It turns out that, in order for \( C' \) to have this property, one needs to require certain properties of the derivation of \( t; I; C; A \vdash_{\text{CFL}} e : \sigma \). Such derivations are called normal derivations in the proof below. However, one can show that normal derivations can be assumed without loss of generality, because an arbitrary derivation in POLYFLOW\_CFL can be normalized.

The strategy of the proof is now the following. We first give a number of technical definitions, in Section A.2. We then establish the property (A.1) in Section A.3 under certain assumptions about the elements of the judgement \( t; I; C; A \vdash_{\text{CFL}} e : \sigma \). The central notion here is that of a normal instantiation context, defined in Section A.2. In Section A.4 we then show that normal derivations satisfy the assumptions of Section A.3, and in Section A.5 we show that normal derivations can always be obtained. Section A.6 concludes the soundness proof by composing these results.

As a technical device for the soundness proof, we extend our type language and the subtype logic with formal joins, as shown in Figure A.1 and Figure A.2. Whenever \( \ell_1, \ldots, \ell_n \) is a series of labels, we can form the formal join \( \bigsqcup_i \ell_i \). Such joins are governed by the rules [LubL] and [LubR] in the extended subtype logic shown in Figure A.2. The extended language allows us to translate fixpoint types in polymorphic recursive typings of system POLYFLOW\_FL into fixpoint types of system POLYFLOW\_copy. A similar method was used by Mossin in [Moss96] to guarantee the existence of type fixpoints in his system.
\[ C \vdash \sigma \leq \sigma \]

\[
\frac{C \vdash \ell_1 \leq \ell_2}{C \vdash \text{int} \ell_1 \leq \text{int} \ell_2} \quad \text{[Int]}
\]

\[
\frac{C \vdash \sigma_1 \leq \sigma' \quad C \vdash \sigma_2 \leq \sigma'_2 \quad C \vdash \ell \leq \ell'}{C \vdash \sigma_1 \times^\ell \sigma_2 \leq \sigma'_1 \times^\ell \sigma'_2} \quad \text{[Pair]}
\]

\[
\frac{C \vdash \sigma_1 \rightarrow^\ell \sigma_2 \leq \sigma'_1 \rightarrow^\ell \sigma'_2}{C \vdash \sigma_1 \rightarrow^\ell \sigma_2 \leq \sigma'_1 \rightarrow^\ell \sigma'_2} \quad \text{[Fun]}
\]

\[
\frac{C \vdash \ell_i \leq \ell, i = 1 \ldots n}{C \vdash \bigwedge_{i=1}^{n} \ell_i \leq \ell} \quad \text{[LubL]}
\]

\[
\frac{i \in \{1, \ldots, n\}}{C \vdash \ell_i \leq \bigwedge_{i=1}^{n} \ell_i} \quad \text{[LubR]}
\]

Figure A.2: Extended subtype relation

A.2 Normal Instantiation Contexts

Definition 1 (Polarized constraint sets) Let \( C \) be a set of flow constraints, let \( \sigma \) be a labelled type and let \( F \subseteq \text{fl}(\sigma) \). We say that \( C \) is polarized with respect to \( \sigma \) and \( F \), written \( C \gg_F \sigma \), iff the following conditions hold for all \( \ell \in F \):

1. whenever \( C \vdash \ell \leq \ell' \) with \( \ell \neq \ell' \), then \( \ell \) occurs negatively in \( \sigma \)
2. whenever \( C \vdash \ell' \leq \ell \) with \( \ell \neq \ell' \), then \( \ell \) occurs positively in \( \sigma \)

\( \Box \)

Definition 2 (Instantiation context) Let \( C \) be a set of flow constraints, let \( I \) be a set of instantiation constraints, let \( A \) be a set of type assumptions, let \( \sigma \) be a labelled type, and let \( \varphi \) be a substitution on labels. The tuple of data \( \langle C, I, A, \sigma, \varphi \rangle \) is called an instantiation context. \( \Box \)

Definition 3 (CFL closure) Let \( C \) be a set of flow constraints and let \( I \) be a set of instantiation constraints. We then construct a new set of flow constraints, called the CFL closure of \( C \) with respect to \( I \) and denoted \( C^I \), defined by

\[ C^I = \{ \ell \leq \ell' \mid I; C \vdash \ell \rightsquigarrow_m \ell' \} \]

That is, \( C^I \) is the relation of matched CFL flow, considered as a flow constraint set. Notice that, by rule [Level] of Figure 2.6, one has \( C \subseteq C^I \) for all \( I \). \( \Box \)

Definition 4 (Extended substitution) Let \( \langle C, I, A, \sigma, \varphi \rangle \) be an instantiation context with \( \text{dom}(\varphi) = \text{gen}(A, \sigma) \). For a label \( \ell \), define the set of labels \( \mathbf{L}(\ell, C, I, A, \sigma) \) by setting

\[ \mathbf{L}(\ell, C, I, A, \sigma) = \{ \ell' \in \text{fl}(A) \cup \text{fl}(\sigma) \mid C^I \vdash \ell' \leq \ell \} \]
We write \( \varphi L(\ell, C, I, A, \sigma) \) to denote the set \( \{ \varphi(\ell') \mid \ell' \in L(\ell, C, I, A, \sigma) \} \). The extended substitution induced by the instantiation context \( \langle C, I, A, \sigma, \varphi \rangle \) is denoted \( \varphi_{C, I, A, \sigma} \). It has domain \( \text{dom}(\varphi_{C, I, A, \sigma}) = \text{gen}(A, \sigma, C) \) and is given by

\[
\varphi_{C, I, A, \sigma}(\ell) = \begin{cases} 
\varphi(\ell) & \text{if } \ell \in \text{gen}(A, \sigma) \\
\bigcup \varphi L(\ell, C, I, A, \sigma) & \text{if } \ell \in \text{gen}(A, \sigma, C) \setminus \text{gen}(A, \sigma)
\end{cases}
\]

\[\square\]

**Definition 5** (Normal instantiation context) An instantiation context \( \langle C, I, A, \sigma, \varphi \rangle \) is called a normal instantiation context at index \( i \), if and only if the following conditions hold:

1. \( \text{dom } \varphi = \text{gen}(A, \sigma) \)
2. \( C \rhd_{\text{dom } \varphi} \sigma \)
3. \( \forall \ell \in \text{fl}(A). I \vdash \varphi \preceq_{\ell_i^i} \ell \) and \( I \vdash \ell \preceq_{\ell_i^i} \ell \)
4. Whenever \( I \vdash \ell_1 \preceq_{\ell_i^i} \ell_2 \) and \( I \vdash \ell_3 \preceq_{\ell_i^i} \ell_4 \) with \( \ell_1 \neq \ell_2 \) and \( \ell_3 \neq \ell_4 \), then \( \ell_2 \neq \ell_3 \)
5. \( \text{fl}(C) = \text{fl}(C') \)

\[\square\]

**Lemma A.1** If \( \langle C, I, A, \sigma, \varphi \rangle \) is a normal instantiation context at index \( i \), then \( C' \rhd_{\text{dom } \varphi} \sigma \).

**Proof:** Consider the first condition for \( C' \rhd_{\text{dom } \varphi} \sigma \) from Definition 1. Suppose that \( \ell \in \text{dom } \varphi \) and \( C' \vdash \ell \leq \ell' \). We then must show that \( \ell \) occurs negatively in \( \sigma \). For this property, it is sufficient to prove that, whenever \( I; C \vdash_{\text{CFL}} \ell \preceq_m \ell' \) with \( \ell \in \text{dom } \varphi \), then \( \ell \) occurs negatively in \( \sigma \). We prove this property by induction on the proof of \( I; C \vdash_{\text{CFL}} \ell \preceq_m \ell' \), by cases over the last rule used in the proof.

Suppose, for the base case, that the last rule used is

\[
\frac{C \vdash \ell \leq \ell'}{I; C \vdash \ell \preceq_m \ell'} \quad \text{[Level]}
\]

Because \( \langle C, I, A, \sigma, \varphi \rangle \) is a normal instantiation context, we have \( C \rhd_{\text{dom } \varphi} \sigma \), and so \( C \vdash \ell \leq \ell' \) implies that \( \ell \) occurs negatively in \( \sigma \).

Suppose that the last rule used is

\[
\frac{I; C \vdash \ell \preceq_m \ell_1 \quad I; C \vdash \ell_1 \preceq_m \ell'}{I; C \vdash \ell \preceq_m \ell'} \quad \text{[Trans]}
\]

By induction hypothesis, \( I; C \vdash \ell \preceq_m \ell_1 \) implies that \( \ell \) occurs negatively in \( \sigma \).

Suppose that the last rule used is

\[
\frac{I \vdash \ell_1 \preceq_{\ell_i^i} \ell \quad I; C \vdash \ell_1 \preceq_m \ell_2 \quad I \vdash \ell_2 \preceq_{\ell_i^i} \ell'}{I; C \vdash \ell \preceq_m \ell'} \quad \text{[Match]}
\]

Suppose that \( \ell_1 \neq \ell \). Now, we have \( \ell \in \text{dom } \varphi \), and therefore (by \( I \vdash \sigma \preceq_{\ell_i^i} \sigma' : \varphi \)) it must be the case that \( I \vdash \ell \preceq_{\ell_i^i} \varphi(\ell) \) with \( \varphi(\ell) \neq \ell \). Therefore, since we also have \( I \vdash \ell_1 \preceq_{\ell_i^i} \ell \), it
follows from the assumption that \( \langle C, I, A, \sigma, \varphi \rangle \) is a normal instantiation context (condition \((v)\) of Definition 5) that \( \ell \neq \ell \), a contradiction. Hence, we must reject the assumption that \( \ell_1 \neq \ell \), so that we now have \( \ell_1 = \ell \). But then the induction hypothesis applied to the premise \( I; C \vdash \ell_1 \Rightarrow_m \ell_2 \) shows that \( \ell_1 = \ell \) occurs negatively in \( \sigma \).

There are no other rules applicable to derive \( I; C \vdash_{CFL} \ell \Rightarrow_m \ell' \), and the proof of the desired property is complete. Proving the second property of Definition 1 is analogous and left out. \( \square \)

**Lemma A.2** Suppose that \( \langle C, I, A, \sigma, \varphi \rangle \) is a normal instantiation context at index \( i \), and suppose furthermore that \( \ell_0 \) and \( \ell_1 \) are labels in \( fl(C) \) such that one of the following conditions is satisfied for each \( j = 0, 1 \):

\[(c1) \ \ell_j \in \text{gen}(A, \sigma), \text{ or} \]
\[(c2) \ \ell_j \notin \text{gen}(A, \sigma, C) \]

Then \( \varphi(\ell_0) \leq \varphi(\ell_1) \in \varphi(C^I) \) implies \( I; C \vdash_{CFL} \varphi(\ell_0) \Rightarrow_m \varphi(\ell_1) \).

**Proof:** Notice that, since \( \ell_0, \ell_1 \in fl(C) \), one has that \( \ell_j \notin \text{gen}(A, \sigma, C) \) implies \( \ell_j \in fl(A) \). Also, since \( \varphi(\ell_0) \leq \varphi(\ell_1) \in \varphi(C^I) \), one has \( \ell_0 \leq \ell_1 \in C^I \). Finally, because \( \langle C, I, A, \sigma, \varphi \rangle \) is normal, Lemma A.1 yields that \( C^I \models_{\text{dom}_\varphi} \sigma \).

Let \( \ell = \varphi(\ell_0) \) and \( \ell' = \varphi(\ell_1) \). We proceed by cases, according to the four possible combinations of the conditions \((c1)\) and \((c2)\).

**Case \((c2) - (c2)\):** \( \ell_0 \notin \text{gen}(A, \sigma, C), \ell_1 \notin \text{gen}(A, \sigma, C) \). Then \( \ell_0 = \ell, \ell_1 = \ell' \), and we have \( \ell_0 \leq \ell' \in C \), from which we get \( I; C \vdash \ell \Rightarrow_m \ell' \) by rule [Level].

**Case \((c1) - (c2)\):** \( \ell_0 \in \text{gen}(A, \sigma), \ell_1 \notin \text{gen}(A, \sigma, C) \). In this case, one has \( \ell_1 = \varphi(\ell_1) = \ell' \). Since \( \ell_0 \in \text{gen}(A, \sigma) \) with \( \ell_0 \leq \ell_1 \in C^I \), it follows from \( C^I \models_{\text{dom}_\varphi} \sigma \) that \( \ell_0 \) occurs negatively in \( \sigma \). It then follows from the assumption that \( \langle C, I, A, \sigma, \varphi \rangle \) is normal (condition \((iii)\) of Definition 5), that we have

\[ I \vdash \ell_0 \lessdot^i_+ \ell \]  \hspace{1cm} \text{(A.2)}

Since \( \ell_1 \in fl(C) \) and \( \ell_1 \notin \text{gen}(A, \sigma, C) \), it follows that \( \ell_1 \in fl(A) \). Therefore, by the assumption that \( \langle C, I, A, \sigma, \varphi \rangle \) is normal (condition \((iv)\) of Definition 5), we have \( I \vdash \ell_1 \lessdot^+_p \ell_1 \) for \( p = + \) and \( p = \pm \). Choosing this relation with \( p = + \), we have \( I \vdash \ell_1 \lessdot^+_p \ell_1 \). Because \( \ell_1 = \ell' \), this yields

\[ I \vdash \ell' \lessdot^+_p \ell \]  \hspace{1cm} \text{(A.3)}

Since \( \ell_0 \leq \ell_1 \in C^I \), it follows that we have \( I; C \vdash \ell_0 \Rightarrow_m \ell_1 \). Because \( \ell' = \ell_1 \), we therefore have

\[ I; C \vdash \ell_0 \Rightarrow_m \ell' \]  \hspace{1cm} \text{(A.4)}

Composing (A.2),(A.3) and (A.4) we get \( I; C \vdash \ell \Rightarrow_m \ell' \), by rule [Match], as desired.

**Case \((c2) - (c1)\):** \( \ell_0 \notin \text{gen}(A, \sigma, C), \ell_1 \in \text{gen}(A, \sigma) \). In this case we have \( \ell_0 = \varphi(\ell_0) = \ell \).

Since \( \ell_1 \in \text{gen}(A, \sigma) \) with \( \ell_0 \leq \ell_1 \in C^I \), it follows from \( C^I \models_{\text{dom}_\varphi} \sigma \) that \( \ell_1 \) occurs positively in \( \sigma \). It then follows from the assumption that \( \langle C, I, A, \sigma, \varphi \rangle \) is normal (condition \((iii)\) of Definition 5), that we have

\[ I \vdash \ell_1 \lessdot^+_p \ell' \]  \hspace{1cm} \text{(A.5)}
Since \( \ell_0 \in fl(C) \) and \( \ell_0 \notin gen(A, \sigma, C) \), it follows that \( \ell_0 \in fl(A) \). Therefore, by the assumption that \( \langle C, I, A, \sigma, \varphi \rangle \) is normal (condition (iv) of Definition 5), we have \( I \vdash \ell_0 \preceq_p \ell_0 \) for \( p = + \) and \( p = \div \). Choosing this relation with \( p = \div \), we have \( I \vdash \ell_0 \preceq_\div \ell_0 \). Because \( \ell_0 = \ell \), this yields

\[
I \vdash \ell \preceq_\div \ell
\]  

(A.6)

Since \( \ell_0 \leq \ell_1 \in C^I \), it follows that we have \( I; C \vdash \ell_0 \sim_m \ell_1 \). Because \( \ell = \ell_0 \), we therefore have

\[
I; C \vdash \ell \sim_m \ell_1
\]  

(A.7)

Composing (A.5),(A.6) and (A.7) we get \( I; C \vdash \ell \sim_m \ell' \), by rule [Match], as desired.

Case (c1) – (c1): \( \ell_0 \in gen(A, \sigma) \) and \( \ell_1 \in gen(A, \sigma) \). Since \( \ell_0 \in dom \varphi \) and \( \ell_1 \in dom \varphi \) with \( \ell_0 \leq \ell_1 \in C^I \), it follows from \( C^I \supset_{dom \varphi} \sigma \) that \( \ell_0 \) occurs negatively in \( \sigma \) and \( \ell_1 \) occurs positively in \( \sigma \). It then follows from the assumption that \( \langle C, I, A, \sigma, \varphi \rangle \) is normal (condition (iii) of Definition 5), that we have

\[
I \vdash \ell_0 \preceq_\div \ell
\]  

(A.8)

and

\[
I \vdash \ell_1 \preceq_\div \ell'
\]  

(A.9)

Since we also have

\[
I; C \vdash \ell_0 \sim_m \ell_1
\]  

(A.10)

by \( \ell_0 \leq \ell_1 \in C^I \), we can apply rule [Match] to (A.8), (A.9) and (A.10) to get \( I; C \vdash_{CFL} \ell \sim_m \ell' \), as desired. \( \square \)

**Corollary 1** Suppose that \( \langle C, I, A, \sigma, \varphi \rangle \) is a normal instantiation context at index \( i \), and let \( \ell_0 \) and \( \ell_1 \) be as stated in Lemma A.2. Then \( \varphi(C^I) \vdash \varphi(\ell_0) \leq \varphi(\ell_1) \) implies \( I; C \vdash_{CFL} \varphi(\ell_0) \sim_m \varphi(\ell_1) \).

**Proof:** By induction on the proof of \( \varphi(C^I) \vdash \varphi(\ell_0) \leq \varphi(\ell_1) \), using Lemma A.2 for the base case, together with reflexivity and transitivity of the relation \( \sim_m \) (using rules [Level] and [Trans] of Figure 2.6) for the inductive cases. \( \square \)

**A.3 Closure Under Substitution**

**Theorem A.3** Let \( \langle C, I, A, \sigma, \varphi \rangle \) be a normal instantiation context at index \( i \), and let \( \widehat{\varphi} = \varphi(\langle C, I, A, \sigma \rangle) \). Then one has

\[
C^I \vdash \widehat{\varphi}(C^I)
\]

**Proof:** We are to show that whenever \( L \leq L' \in \widehat{\varphi}(C^I) \), then \( C^I \vdash L \leq L' \). So assume, for arbitrary \( L, L' \) that \( L \leq L' \in \widehat{\varphi}(C^I) \). Then, for some \( \ell_0, \ell_1 \in fl(C^I) = fl(C) \), we have \( \ell_0 \leq \ell_1 \in C^I \) and \( \widehat{\varphi}(\ell_0) = L \) and \( \widehat{\varphi}(\ell_1) = L' \). We can assume \( \ell_0 \neq \ell_1 \), since otherwise we are done. For each label \( \ell_j, j = 0,1 \), there are three disjoint cases to consider:
(c1) \( \ell_j \in \text{gen}(A, \sigma) \)

(c2) \( \ell_j \in \text{gen}(A, \sigma, C) \setminus \text{gen}(A, \sigma) \)

(c3) \( \ell_j \notin \text{gen}(A, \sigma, C) \)

This yields 9 possible cases for \( \ell_0 \) and \( \ell_1 \). We prove \( C^I \vdash L \preceq L' \) by considering each of these 9 possible cases. Notice that one has \( \text{gen}(A, \sigma) = \text{dom}(\varphi) \), \( \text{gen}(A, \sigma, C) = \text{dom}(\widehat{\varphi}) \) and hence \( \text{gen}(A, \sigma, C) \setminus \text{gen}(A, \sigma) = \text{dom}(\widehat{\varphi}) \setminus \text{dom}(\varphi) \).

First consider all cases where \( \ell_0 \) and \( \ell_1 \) are either in \( \text{gen}(A, \sigma) = \text{dom}(\varphi) \) or not in \( \text{gen}(A, \sigma, C) = \text{dom}(\widehat{\varphi}) \). These are just the combinations of the cases (c1) and (c3) above, and in these cases we have \( \widehat{\varphi} = \varphi \), and Lemma A.2 gives the conclusion

\[ I; C \vdash L \sim_m L' \]

in all of these cases. It then follows, by definition of \( C^I \), that \( C^I \vdash L \preceq L' \) in all these cases.

We therefore only need to consider the remaining 5 combinations of (c1), (c2), (c3) which are not combinations of (c1) and (c3).

Case (c1) - (c2): \( \ell_0 \in \text{gen}(A, \sigma), \ell_1 \in \text{gen}(A, \sigma, C) \setminus \text{gen}(A, \sigma) \). In this case we have by Definition 4 that

\[ L' = \widehat{\varphi}(\ell_1) = \bigsqcup \varphi L(\ell_1, C, I, A, \sigma) \] \hspace{1cm} (A.11)

and

\[ L = \varphi(\ell_0) \] \hspace{1cm} (A.12)

Since \( \ell_0 \in \text{gen}(A, \sigma) \), we have \( \ell_0 \in fl(\sigma) \). Because \( \ell_0 \leq \ell_1 \in C^I \), we can conclude \( \ell_0 \in L(\ell_1, C, I, A, \sigma) \), by transitivity of \( \leq \). It follows that

\[ \varphi(\ell_0) \in \varphi L(\ell_1, C, I, A, \sigma) \] \hspace{1cm} (A.13)

By (A.13) together with rule [LubR], one has \( \vdash \varphi(\ell_0) \leq \bigsqcup \varphi L(\ell_1, C, I, A, \sigma) \). Using (A.11) and (A.12), we can conclude \( \vdash \widehat{\varphi}(\ell_0) \leq \widehat{\varphi}(\ell_1) \), and hence \( C^I \vdash L \preceq L' \) as desired.

Case (c2) - (c1): \( \ell_0 \in \text{gen}(A, \sigma, C) \setminus \text{gen}(A, \sigma) \). In this case one has

\[ L = \widehat{\varphi}(\ell_0) = \bigsqcup \varphi L(\ell_0, C, I, A, \sigma) \] \hspace{1cm} (A.14)

and

\[ L' = \varphi(\ell_1) \] \hspace{1cm} (A.15)

Since \( \ell_1 \in \text{gen}(A, \sigma) \), one has \( \ell_1 \in fl(\sigma) \). Because \( \ell_0 \leq \ell_1 \in C^I \), we have

\[ \forall \ell'' \in L(\ell_0, C, I, A, \sigma), C^I \vdash \ell'' \leq \ell_1 \] \hspace{1cm} (A.16)

Now, let \( \ell'' \) be an arbitrary label in \( L(\ell_0, C, I, A, \sigma) \). Since \( \vdash \) is preserved under substitutions, we can conclude \( \varphi(C^I) \vdash \varphi(\ell'') \leq \varphi(\ell_1) \) from (A.16). By definition of \( L(\ell_0, C, I, A, \sigma) \), one has \( \ell'' \in fl(A) \cup fl(\sigma) \), and therefore either \( \ell'' \in \text{gen}(A, \sigma) \) or \( \ell'' \notin \text{gen}(A, \sigma, C) \). Moreover, \( \ell_1 \in \text{gen}(A, \sigma) \). Corollary 1 is therefore applicable, and allows us to conclude

\[ \forall \ell'' \in L(\ell_0, C, I, A, \sigma), I; C \vdash_{\text{cfl}} \varphi(\ell'') \sim_m \varphi(\ell_1) \] \hspace{1cm} (A.17)
From (A.17) we can conclude that $\varphi(\ell'') \leq \varphi(\ell_1) \in C^I$ for all $\ell'' \in L(\ell_0, C, I, A, \sigma)$. By rule [LubL] we then get $C^I \vdash \bigcup \varphi L(\ell_0, C, I, A, \sigma) \leq \varphi(\ell_1)$, hence $C^I \vdash L \leq L'$ follows from (A.14) and (A.15), thereby proving the theorem in this case.

**Case (c2) – (c2):** $\ell_0 \in \text{gen}(A, \sigma) \setminus \text{gen}(A, \sigma), \ell_1 \in \text{gen}(A, \sigma) \setminus \text{gen}(A, \sigma)$. In this case one has

$$L = \hat{\varphi}(\ell_0) = \bigcup \varphi L(\ell_0, C, I, A, \sigma) \quad (A.18)$$

and

$$L' = \hat{\varphi}(\ell_1) = \bigcup \varphi L(\ell_1, C, I, A, \sigma) \quad (A.19)$$

Now, for all $\ell'' \in L(\ell_0, C, I, A, \sigma)$ one has $C^I \vdash \ell'' \leq \ell_0$. Hence, using $\ell_0 \leq \ell_1 \in C^I$, one has $C^I \vdash \ell'' \leq \ell_1$ for all $\ell'' \in L(\ell_0, C, I, A, \sigma)$. It follows that $L(\ell_0, C, I, A, \sigma) \subseteq L(\ell_1, C, I, A, \sigma)$, and therefore we can conclude

$$\varphi L(\ell_0, C, A, \sigma) \subseteq \varphi L(\ell_1, C, A, \sigma) \quad (A.20)$$

From (A.20), together with an application of rule [LubR] followed by an application of rule [LubL], we get

$$\vdash \bigcup \varphi L(\ell_0, C, I, A, \sigma) \leq \bigcup \varphi L(\ell_1, C, I, A, \sigma) \quad (A.21)$$

But (A.21), (A.18) and (A.19) together show that $\vdash L \leq L'$, hence also $C^I \vdash L \leq L'$, as desired.

**Case (c3) – (c2):** $\ell_0 \not\in \text{gen}(A, \sigma, C), \ell_1 \in \text{gen}(A, \sigma, C) \setminus \text{gen}(A, \sigma)$. In this case one has

$$L = \hat{\varphi}(\ell_0) = \varphi(\ell_0) = \ell_0 \quad (A.22)$$

and

$$L' = \hat{\varphi}(\ell_1) = \bigcup \varphi L(\ell_1, C, I, A, \sigma) \quad (A.23)$$

Notice that, because $\ell_0 \not\in \text{gen}(A, \sigma, C)$, we must have $\ell_0 \in f(A)$. Since $\ell_0 \leq \ell_1 \in C^I$, it follows that $\ell_0 \in L(\ell_1, C, I, A, \sigma)$, hence $\varphi(\ell_0) \in \varphi L(\ell_1, C, I, A, \sigma)$, and therefore $\vdash \varphi(\ell_0) \leq \bigcup \varphi L(\ell_1, C, I, A, \sigma)$ by rule [LubR]. Using (A.22) and (A.23) we can conclude $\vdash L \leq L'$, hence $C^I \vdash L \leq L'$, as desired.

**Case (c2) – (c3):** $\ell_0 \in \text{gen}(A, \sigma, C) \setminus \text{gen}(A, \sigma), \ell_1 \not\in \text{gen}(A, \sigma, C)$. In this case one has

$$L = \hat{\varphi}(\ell_0) = \bigcup \varphi L(\ell_0, C, I, A, \sigma) \quad (A.24)$$

and

$$L' = \hat{\varphi}(\ell_1) = \varphi(\ell_1) = \ell_1 \quad (A.25)$$

Since $\ell_0 \leq \ell_1 \in C^I$, we have $C^I \vdash \ell'' \leq \ell_1$ for all $\ell'' \in L(\ell_0, C, I, A, \sigma)$. Since $\vdash$ is preserved under substitution, it follows that

$$\forall \ell'' \in L(\ell_0, C, I, A, \sigma), \varphi(C^I) \vdash \varphi(\ell'') \leq \varphi(\ell_1) \quad (A.26)$$
Let $\ell''$ be an arbitrary label in $L(\ell_0, C, I, A, \sigma)$. Then $\ell'' \in fl(A) \cup fl(\sigma)$ by definition. It follows that either $\ell'' \in gen(A, \sigma)$ or $\ell'' \in gen(A, \sigma, C) \setminus gen(A, \sigma)$. In either case, Corollary 1 is applicable to $\ell''$ and $\ell_1$ because $\ell_1 \notin gen(A, \sigma, C)$. From (A.26), we then conclude from Corollary 1 that

$$\forall \ell'' \in L(\ell_0, C, I, A, \sigma). I; C \vdash_{\text{CFL}} \varphi(\ell'') \sim_m \varphi(\ell_1)$$

(A.27)

In turn, it follows from (A.27) that $\varphi(\ell'') \leq \varphi(\ell_1) \in C^I$ for all $\ell'' \in L(\ell_0, C, I, A, \sigma)$. By rule [LubL] we can then conclude that $C^I \vdash \bigcup \varphi L(\ell_0, C, I, A, \sigma) \leq \varphi(\ell_1)$, thereby showing $C^I \vdash \hat{\varphi}(\ell_0) \leq \hat{\varphi}(\ell_1)$ and hence (using (A.25) and (A.26)) we get $C^I \vdash L \leq L'$, as desired. □

### A.4 Normal Derivations

**Definition 6 (Normal derivation)** Let $D$ be a derivation (regarded as a proof tree) in $\text{POLYFLOW}_{\text{CFL}}$. Each application within $D$ of rule [Inst] of the form

$$I \vdash \sigma \triangleleft_{I^*} \sigma': \varphi \quad \text{dom } \varphi = \ell^* \quad I \vdash s \triangleleft_{I^*} s_i \quad I \vdash s_i \triangleleft_{I^*} s_i \quad \text{[Inst]}$$

$$t; I; C; A; f : (\forall \ell, \sigma, s) \vdash_{\text{CFL}} f : \sigma'$$

is uniquely determined by the index $i$. Moreover, for every let- or letrec-bound variable $f$ we can assume (by suitably renaming variables) that there is a unique application of rule [Let] or [Rec], respectively, in which the symbol $f$ gets bound. The second premise of such a rule application of the form

$$t; I; C; A; f : (\forall \ell, \sigma, s) \vdash_{\text{CFL}} e : \sigma_2$$

is referred to as the **binding premise** for $f$ within $D$. The environment $A$ in this rule application is denoted $\text{env}(D, f)$ (notice that one has $\ell = \text{gen}(\text{env}(D, f), \sigma_1)$ in the binding premise for $f$).

The first premise of an application of rule [Let] or [Rec] (that is, the premise which is not the binding premise) is referred to as the **minor premise** of the rule application.

An instantiation site within $D$ given by the symbol $f^*$ uniquely determines an application of rule [Inst] as shown above. This site determines the set of data $I, C, f, \sigma, \varphi$. It thereby determines the instantiation context $(C, I, \text{env}(D, f), \sigma, \varphi)$, which is called the **instantiation context determined by this application of rule [Inst]**. We also refer to the context $(C, I, \text{env}(D, f), \sigma, \varphi)$ as the **instantiation context determined by index $i$**, and we may denote it $\text{Inston}(D, i)$.

Let $\text{Inston}(D)$ denote the set of all instantiation contexts determined by applications of rule [Inst] in $D$. We say that a derivation $D$ is a **normal derivation** if and only if every instantiation context in $\text{Inston}(D)$ is normal (according to Definition 5). □

**Definition 7 (Translation on type assumptions)** Let $A$ be a set of type assumptions, $C$ a set of flow constraints and $I$ a set of instantiation constraints. For a quantified type of the form $\forall \ell, \sigma$, we define the qualified type $(\forall \ell, \sigma)^{(I, C, A)}$ by setting

$$(\forall \ell, \sigma)^{(I, C, A)} = \forall \ell. C^I \Rightarrow \sigma \text{ with } \ell = \text{gen}(A, \sigma, C)$$

Let $D$ be a derivation of the judgement

$$t; I; C; A' \vdash_{\text{CFL}} e : \sigma$$

We translate a set of type assumptions $A$ into $A^D$ given by
\( \emptyset^D = \emptyset \)

\((A, x : \sigma)^D = A^D, x : \sigma \)

\((A, f : (\forall \ell. \sigma, t))^D = A^D, f : (\forall \ell. \sigma)^{I, C, \text{env}(D, f)} \)

\( \square \)

**Definition 8** (Extension of instantiation sets) Let \( D \) be a derivation in \( \text{POLYFLOW}_{\text{CFL}} \) of the judgement \( t; I; C; A \vdash_{\text{CFL}} e : \sigma \). We then define an extended set of instantiation constraints \( I^D \):

\[
I^D = I \cup \{ \ell \preceq^i \varphi_{(C, I, A, \sigma)}(\ell) \mid \langle C, A, \sigma, \varphi \rangle = \text{Instcon}(D, i) \}
\]

Here, \( \varphi_{(C, I, A, \sigma)} \) is the extended substitution induced by the instantiation context \( \langle C, I, A, \sigma \rangle \), according to Definition 4. \( \square \)

**Lemma A.4** (Translation of normal derivations) Suppose that

\( t; I; C; A \vdash_{\text{CFL}} e : \sigma \)

is derivable by a normal derivation \( D \) in system \( \text{POLYFLOW}_{\text{CFL}} \). Then the judgement

\( C^I; I^D; C^I; A^D \vdash_{\text{cp}} e : \sigma \)

is derivable in system \( \text{POLYFLOW}_{\text{copy}} \).

**Proof:** Let \( D \) be a normal derivation in system \( \text{POLYFLOW}_{\text{CFL}} \) of the judgement \( t; I; C; A \vdash_{\text{CFL}} e : \sigma \). We translate every \( \text{POLYFLOW}_{\text{CFL}} \)-judgement of the form

\( t_0; I_0; C_0; A_0 \vdash_{\text{CFL}} e_0 : \sigma_0 \)

in \( D \) into the \( \text{POLYFLOW}_{\text{copy}} \)-judgement

\( C^{I_0}_0; I^D_0; C^{I_0}_0; A^D \vdash_{\text{cp}} e_0 : \sigma_0 \)

The derivation \( D \) thereby translates into a series of judgements, \( D' \), in \( \text{POLYFLOW}_{\text{copy}} \). One can then verify, by induction on the derivation \( D \), that the translated sequence of judgements \( D' \) constitutes a valid derivation in \( \text{POLYFLOW}_{\text{copy}} \). The proof is by cases over the last rule applied in \( D \). We will do a few illustrative cases:

Suppose that the last rule used in \( D \) is \([\text{Inst}]\), of the form

\[
\frac{I \vdash \sigma \preceq^i \sigma' : \varphi \quad \text{dom } \varphi = \bar{\ell} \quad I \vdash s \preceq^i s \quad I \vdash s \preceq^i s}{t; I; C; A, f : (\forall \ell. \sigma, s) \vdash_{\text{CFL}} f^i : \sigma'}
\]

Let \( \langle I, C, A', \varphi, \sigma' \rangle \) be the instantiation context \( \text{Instcon}(D, i) = \langle I, C, \text{env}(D, f), \varphi, \sigma \rangle \), and let \( \widehat{\varphi} = \varphi_{(C, I, A', \varphi, \sigma)} \). It follows from the premise \( I \vdash \sigma \preceq^i \sigma' : \varphi \) together with the definition of \( I^D \) (Definition 8) that we have

\[
I^D \vdash \sigma \preceq^i \sigma' : \widehat{\varphi}
\]
Also, since the context \( \langle I, C, A', \varphi, \sigma \rangle \) is normal by assumption, Theorem A.3 shows that we have

\[
C^I \vdash \hat{\varphi}(C^I)
\]  
(A.29)

Finally, by definition of \( A^D \) (Definition 7) we have

\[
(A, f : (\forall \ell. \sigma, s))^D = A^D, f : \forall \hat{\ell}. C^I \Rightarrow \sigma
\]  
(A.30)

with \( \hat{\ell} = \text{gen}(\text{env}(A', f)) \). It follows from (A.28), (A.29) and (A.30) that the following application of rule \([\text{Inst}]\) is valid in \( \text{POLYFLOW}_{\text{copy}} \):

\[
\begin{array}{c}
I^D \vdash \sigma <_C \sigma' \quad C^I \vdash \hat{\varphi}(C^I) \\
\quad \text{dom } \hat{\varphi} = \hat{\ell}
\end{array}
\quad \frac{C^I ; I^D ; C^I ; A^D, f : \forall \hat{\ell}. C^I \Rightarrow \sigma \vdash_{cp} f^I : \sigma'}{[\text{Inst}]}
\]

and thereby proving the lemma in this case.

Suppose that the last rule used in \( D \) is \([\text{Sub}]\), of the form

\[
\begin{array}{c}
t ; I ; C ; A \vdash_{\text{CFL}} e : \sigma
\end{array}
\quad \frac{C \vdash \sigma < \sigma'}{[\text{Sub}]}
\]

By induction, we have \( C^I ; I^D ; C^I ; A^D \vdash_{cp} e : \sigma \) derivable in \( \text{POLYFLOW}_{\text{copy}} \). Since \( C \subseteq C^I \), one has \( C^I \vdash \sigma \leq \sigma' \) by \( C \vdash \sigma \leq \sigma' \). Hence, rule \([\text{Sub}]\) is applicable in \( \text{POLYFLOW}_{\text{copy}} \), and it yields the conclusion yields \( C^I ; I^D ; C^I ; A^D \vdash_{cp} e : \sigma' \), thereby proving the lemma in this case.

Suppose that the last rule used in \( D \) is an application of the rule \([\text{Let}]\), of the form

\[
\begin{array}{c}
t ; I ; C ; A \vdash_{\text{CFL}} e_1 : \sigma_1 \\
\quad \tilde{e} = \text{gen}(A, \sigma_1)
\end{array}
\quad \frac{t ; I ; C ; A \vdash f : (\forall \ell. \sigma_1, t) \vdash_{\text{CFL}} e_2 : \sigma_2}{[\text{Let}]}
\]

By induction, we know that

\[
C^I ; I^D ; C^I ; A^D \vdash_{cp} e_1 : \sigma_1
\]

is derivable, and

\[
\frac{C^I ; I^D ; C^I ; A^D, f : \forall \hat{\ell}. C^I \Rightarrow \sigma_1 \vdash_{cp} e_2 : \sigma_2}{C^I \vdash C^I \quad C^I \subseteq C^I}
\]

is derivable, with \( \hat{\ell} = \text{gen}(\text{env}(D, f), \sigma_1) \). Now, by definition of \( \text{env}(D, f) \) (Definition 7) and \( A^D \) (Definition 6), we can conclude that \( A = \text{env}(D, f) \). Moreover, because \( D \) is normal, we have \( f(C) = f(C^I) \). It follows that we can write \( \hat{\ell} = \text{gen}(A, \sigma_1, C) \). Therefore, the following application of the \([\text{Let}]\) rule is valid in \( \text{POLYFLOW}_{\text{copy}} \):

\[
\begin{array}{c}
C^I ; I^D ; C^I ; A^D \vdash_{cp} e_1 : \sigma_1
\end{array}
\quad \frac{C^I ; I^D ; C^I ; A^D \vdash_{cp} e_2 : \sigma_2 \quad C^I \subseteq C^I}{C^I \vdash \hat{\ell} \subseteq \text{gen}(A, \sigma_1, C^I)}
\quad \frac{C^I ; I^D ; C^I ; A^D \vdash_{cp} f = e_1 \text{ in } e_2 : \sigma_2}{[\text{Let}]}
\]

thereby proving the lemma in this case.
Suppose that the last rule used in \( D \) is an application of the rule [Ret], of the form

\[
\begin{align*}
    t; I; C; A; f : (\forall \bar{\ell}. \sigma_1, t) &\vdash_{CFL} e_1 : \sigma_1 \\
    t; I; C; A; f : (\forall \bar{\ell}. \sigma_1, t) &\vdash_{CFL} e_2 : \sigma_2 \\
    t; I; C; A &\vdash_{CFL \text{ letrec} f = e_1 \text{ in } e_2 : \sigma_2} \quad \text{[Rec]}
\end{align*}
\]

By induction, we know that

\[C^I ; I^D; C^I; A^D, f : \forall \bar{\ell}. C^I \Rightarrow \sigma_1 \vdash_{cp} e_1 : \sigma_1\]

is derivable in POLYFLOW_{copy}, and

\[C^I ; I^D; C^I; A^D, f : \forall \bar{\ell}. C^I \Rightarrow \sigma_1 \vdash_{cp} e_2 : \sigma_2\]

is derivable in POLYFLOW_{copy}, with \( \ell' = \text{gen}(\text{env}(D, f), \sigma_1) \). Now, by definition of \( \text{env}(D, f) \) (Definition 7) and \( A^D \) (Definition 6), we can conclude that \( A = \text{env}(D, f) \). Moreover, because \( D \) is normal, we have \( f \ell (C) = f \ell (C') \). It follows that we can write \( \bar{\ell} = \text{gen}(A, \sigma_1, C) \). Therefore, the following application of the [Let] rule is invalid in POLYFLOW_{copy}:

\[
\begin{align*}
    &C^I ; I^D; C^I; A^D, f : \forall \bar{\ell}. C^I \Rightarrow \sigma_1 \vdash_{cp} e_1 : \sigma_1 \\
    &C^I ; I^D; C^I; A^D, f : \forall \bar{\ell}. C^I \Rightarrow \sigma_1 \vdash_{cp} e_2 : \sigma_2 \\
    &C^I \subseteq C^I \\
    &\ell' \subseteq \text{gen}(A, \sigma_1, C^I) \vdash \tau(s) : \sigma_1 \\
    &\vdash_{\tau} : \sigma_1 \text{ in } e_2 : \sigma_2 \quad \text{[Rec]}
\end{align*}
\]

thereby proving the lemma in this case.

The remaining cases are obvious and are left out. \( \square \)

### A.5 Existence Of Normal Derivations

**Definition 9** (Label closure) For labeled types \( \sigma_1, \sigma_2 \) we define the label closure of the inequality \( \sigma_1 \leq \sigma_2 \), denoted \( \text{ld}(\sigma_1 \leq \sigma_2) \), to be the set of flow constraints implied by the constraint \( \sigma_1 \leq \sigma_2 \), as follows:

- \( \text{ld}(\text{int}^{\ell_1} \leq \text{int}^{\ell_2}) = \{\ell_1 \leq \ell_2\} \)
- \( \text{ld}(\sigma_1 \times^{\ell_1} \sigma_2 \leq \sigma_3 \times^{\ell_2} \sigma_4) = \{\ell_1 \leq \ell_2\} \cup \text{ld}(\sigma_1 \leq \sigma_3) \cup \text{ld}(\sigma_2 \leq \sigma_4) \)
- \( \text{ld}(\sigma_1 \rightarrow^{\ell_1} \sigma_2 \leq \sigma_3 \rightarrow^{\ell_2} \sigma_4) = \{\ell_1 \leq \ell_2\} \cup \text{ld}(\sigma_3 \leq \sigma_1) \cup \text{ld}(\sigma_2 \leq \sigma_4) \)

Notice that one has

\[\text{ld}(\sigma_1 \leq \sigma_2) \vdash \sigma_1 \leq \sigma_2\]

\( \square \)

**Lemma A.5** (Existence of normal derivations) For every derivation \( D \) of a judgement

\[t; I; C; A \vdash_{CFL} e : \sigma\]

in POLYFLOW_{CFL} there exist \( C' \) and \( I' \) and a normal derivation \( D_N \) of

\[t; I'; C'; A \vdash_{CFL} e : \sigma\]

such that for all \( \ell, \ell' \) in \( f\ell(e) \) one has

\[I'; C' \vdash \ell \rightsquigarrow_{m} \ell' \Rightarrow I; C \vdash \ell \rightsquigarrow_{m} \ell'\]

71
The proof is in 3 steps. We first (step I) introduce transformations on derivations, which transforms a given derivation $\mathcal{D}$ into a new derivation $\mathcal{D}_N$. We then show (step II) that the resulting derivation $\mathcal{D}_N$ is indeed normal. Finally (step III), we show that the property
\[
I;C \vdash \ell \rightsimeq_m \ell' \Rightarrow I;C \vdash \ell \rightsimeq_m \ell'
\]
holds.

We assume w.l.o.g. that the given derivation $\mathcal{D}$ satisfies the following binding properties:

- Any two program variables bound by distinct let- or letrec-bindings are distinct, and
- Whenever $f : (\forall \ell. \sigma, t)$ and $g : (\forall \ell'. \sigma', t')$ are two distinct type assumptions occurring in distinct binding premises (Definition 6) of $\mathcal{D}$, then the labels in $\ell$ and $\ell'$ are disjoint.
- Whenever $\varphi_1$ and $\varphi_2$ are substitutions used at any two applications of rule [Inst] in $\mathcal{D}$, one has $\text{dom } \varphi_1 \cap \text{dom } \varphi_2 = \emptyset$.

These properties can always be obtained by renaming let- or letrec-bound program variables and by renaming bound type labels.

Step I. Let a derivation $\mathcal{D}$ of the judgement $t;I;C;A \vdash_{\text{CFL}} e : \sigma$ be given in POLY-FLOW$_{\text{CFL}}$. To obtain $\mathcal{D}_N$ from $\mathcal{D}$, we perform local transformations on every application of rule [Let] and [Rec] in $\mathcal{D}$, as described next.

To describe the transformations of derivations, we need a few technical preliminaries.

For each let- or letrec-bound variable $f$ in $\mathcal{D}$, let $TyBind(\mathcal{D}, f) = \sigma$, where $\sigma$ is the uniquely determined type such that the binding $f : (\forall \ell. \sigma, t)$ occurs in a binding premise within $\mathcal{D}$, and let $LabBind(\mathcal{D}, f)$ denote the set of labels in $\ell$, where $\ell$ is the vector of labels bound in that binding premise. Notice that the binding properties assumed for $\mathcal{D}$ ensure that one has
\[
LabBind(\mathcal{D}, f) \cap LabBind(\mathcal{D}, g) = \emptyset
\]
for any $f$ and $g$ occurring in distinct binding premises in $\mathcal{D}$. Fix a let- or letrec-bound variable $f$ in $\mathcal{D}$. Let $\psi_f$ be a renaming on $LabBind(\mathcal{D}, f)$, mapping the labels in $LabBind(\mathcal{D}, f)$ to distinct labels which are to be kept fresh throughout $\mathcal{D}_N$ such that for any substitutions $\varphi_1, \varphi_2$ applied at any instantiation sites in $\mathcal{D}_N$, one has:
\[
\text{dom } \varphi_1 \cap \text{dom } \varphi_2 = \emptyset \quad (A.31)
\]
and
\[
\text{dom } \varphi_1 \cap \text{ran } \varphi_2 = \emptyset \quad (A.32)
\]
where we set $\text{ran } \varphi = \{ \varphi(\ell) \mid \ell \in \text{dom } \varphi \}$. Using our assumption that $\mathcal{D}$ satisfies the binding properties mentioned above, it is clear that $\psi_f$ can be chosen in such a way that these conditions are satisfied.

For each $\sigma \in TyBind(\mathcal{D}, f)$, let $\sigma^f = \psi_f(\sigma)$ and define the constraint set $C^f$ by
\[
C^f = C \cup \text{let}(\sigma \leq \sigma^f)
\]
and define the set of instantiations constraints \( I' \) by

\[
I' = \{ \psi_j(\ell) \geq_p \ell' \mid \ell \leq_p \ell' \in I \}
\]

**Transformation of applications of rule \([\text{Let}]\).** Consider an untransformed application of rule \([\text{Let}]\) within \( D \) of the form

\[
\begin{align*}
t;I;C;A \vdash_{\text{CFL}} e_1 : \sigma_1 \\
\ell = \text{gen}(A, \sigma_1) \\
\sigma_1 \vdash_{\text{CFL}} e_2 : \sigma_2 \\
\frac{}{t;I;C;A \vdash_{\text{CFL}} \text{let } f = e_1 \text{ in } e_2 : \sigma_2} \quad \text{[Let]}
\end{align*}
\]

This application of rule \([\text{Let}]\) gets transformed into the following series of judgements:

\[
\begin{align*}
t;I'_1;C'\ell;A \vdash_{\text{CFL}} e_1 : \sigma_1 \\
= \text{gen}(A, \sigma_1') \\
C' \vdash_{\text{CFL}} \sigma_1 \leq \sigma_1' \quad \text{[Sub]} \\
\frac{}{t;I'_1;C'\ell;A \vdash_{\text{CFL}} e_1 : \sigma_1'} \\
\frac{}{t;I'_1;C'\ell;A \vdash_{\text{CFL}} \text{let } f = e_1 \text{ in } e_2 : \sigma_2} \quad \text{[Let]}
\end{align*}
\]

where

\[
J = t;I'_1;C'\ell;A, f : (\forall x. \sigma_1', t) \vdash_{\text{CFL}} e_2 : \sigma_2
\]

and \( \ell = \text{gen}(A, \sigma_1') \). This series of judgements enters into a valid subderivation of \( \text{POLYFLOW}_{\text{CFL}} \). To see this, first consider that we have

\[
\begin{align*}
t;I'_1;C'\ell;A \vdash_{\text{CFL}} e_1 : \sigma_1 \\
C' \vdash_{\text{CFL}} \sigma_1 \leq \sigma_1' \quad \text{[Sub]} \\
\frac{}{t;I'_1;C'\ell;A \vdash_{\text{CFL}} e_1 : \sigma_1'}
\end{align*}
\]

because \( \text{let}(\sigma_1 \leq \sigma_1') \subseteq C' \) by definition of \( C' \). Secondly, since by assumption we can derive

\[
t;I;C;A, f : (\forall x. \sigma_1, t) \vdash_{\text{CFL}} e_2 : \sigma_2
\]

we can also derive

\[
t;I'_1;C'\ell;A \vdash_{\text{CFL}} e_2 : \sigma_2
\]

by renaming of bound variables: any instantiation of the type of \( f \) used to type \( e_2 \) in the derivation of the former judgement can be performed on the type of \( f \) in the latter judgement to type \( e_2 \) in the same way. It follows that we can apply the \([\text{Let}]\) rule as follows:

\[
\begin{align*}
t;I'_1;C'\ell;A \vdash_{\text{CFL}} e_1 : \sigma_1' \\
\ell = \text{gen}(A, \sigma_0) \\
\sigma_1 \vdash_{\text{CFL}} e_2 : \sigma_2 \\
\frac{}{t;I'_1;C'\ell;A \vdash_{\text{CFL}} \text{let } f = e_1 \text{ in } e_2 : \sigma_2} \quad \text{[Let]}
\end{align*}
\]

We can now compose the application of \([\text{Sub}]\) with the application of \([\text{Let}]\) to obtain a valid subproof of

\[
t;I'_1;C'\ell;A, f : (\forall x. \sigma_1', t) \vdash_{\text{CFL}} e_2 : \sigma_2
\]
**Transformation of applications of rule [Rec]**. Consider an untransformed application of rule [Rec] within \( \mathcal{D} \) of the form

\[
\begin{align*}
  t; I; C; A, f : (\forall \bar{\theta}. \sigma_1, t) \vdash_{\text{CFL}} e_1 : \sigma_1 \\
  \bar{\theta} = \text{gen}(A, \sigma_1)
\end{align*}
\]

This application of [Rec] gets transformed into the following series of judgements:

\[
\begin{align*}
  & t; I'; C'; A, f : (\forall \bar{\theta}. \sigma'_1, t) \vdash_{\text{CFL}} e_1 : \sigma_1 \\
  & C' \vdash \sigma_1 \leq \sigma'_1 \quad \text{[Sub]} \\
  \hline
  & t; I'; C'; A, f : (\forall \bar{\theta}. \sigma'_1, t) \vdash_{\text{CFL}} e_1 : \sigma'_1 \\
  \hline
  & t; I'; C'; A \vdash_{\text{CFL letrec}} f = e_1 \text{ in } e_2 : \sigma_2 \quad \text{[Rec]}
\end{align*}
\]

where

\[
\text{J} = t; I'; C'; A, f : (\forall \bar{\theta}. \sigma'_1, t) \vdash_{\text{CFL}} e_2 : \sigma_2
\]

and \( \bar{\theta} = \text{gen}(A, \sigma'_1) \). This series of judgements enters into a valid subderivation of POLYFLOWCFL. To see this, first consider that we have

\[
\begin{align*}
  t; I'; C'; A, f : (\forall \bar{\theta}. \sigma'_1, t) \vdash_{\text{CFL}} e_1 : \sigma_1
\end{align*}
\]

derivable, because (by assumption) we can derive

\[
\begin{align*}
  t; I; C; A, f : (\forall \bar{\theta}. \sigma_1, t) \vdash_{\text{CFL}} e_1 : \sigma_1
\end{align*}
\]

The derivability of the former judgement follows from the derivability of the latter by renaming bound variables, as argued previously (in the case of [Let], above). We then have

\[
\begin{align*}
  & t; I'; C'; A, f : (\forall \bar{\theta}. \sigma'_1, t) \vdash_{\text{CFL}} e_1 : \sigma_1 \\
  & C' \vdash \sigma_1 \leq \sigma'_1 \quad \text{[Sub]} \\
  \hline
  & t; I'; C'; A, f : (\forall \bar{\theta}. \sigma'_1, t) \vdash_{\text{CFL}} e_1 : \sigma'_1 \\
  \hline
  & t; I'; C'; A \vdash_{\text{CFL letrec}} f = e_1 \text{ in } e_2 : \sigma_2 \quad \text{[Rec]}
\end{align*}
\]

because \( \text{let}(\sigma_1 \leq \sigma'_1) \subseteq C' \) by definition of \( C' \). Finally, since by assumption we can derive

\[
\begin{align*}
  t; I; C; A, f : (\forall \bar{\theta}. \sigma_1, t) \vdash_{\text{CFL}} e_2 : \sigma_2
\end{align*}
\]

we can also derive

\[
\begin{align*}
  t; I'; C'; A, f : (\forall \bar{\theta}. \sigma'_1, t) \vdash_{\text{CFL}} e_2 : \sigma_2
\end{align*}
\]

by renaming of bound variables. We can then apply the [Rec] rule as follows:

\[
\begin{align*}
  & t; I'; C'; A, f : (\forall \bar{\theta}. \sigma'_1, t) \vdash_{\text{CFL}} e_0 : \sigma'_1 \\
  \bar{\theta} = \text{gen}(A, \sigma'_1)
\end{align*}
\]

\[
\begin{align*}
  & t; I'; C'; A, f : (\forall \bar{\theta}. \sigma'_1, t) \vdash_{\text{CFL}} e_2 : \sigma_2 \\
  \hline
  & t; I'; C'; A \vdash_{\text{CFL letrec}} f = e_1 \text{ in } e_2 : \sigma_2 \quad \text{[Rec]}
\end{align*}
\]

We can now compose the application of [Sub] with the application of [Rec] to obtain a valid subproof of

\[
\begin{align*}
  t; I'; C'; A, f : (\forall \bar{\theta}. \sigma'_1, t) \vdash_{\text{CFL}} e_2 : \sigma_2
\end{align*}
\]
By repeated application of the two proof transformation we can construct a transformed derivation $D'$ of

$$t; I^*; C^*; A \vdash_{CFL} e : \sigma$$

Finally, in order to turn $D'$ into a normal derivation $D_N$, we add the trivial inequality $\ell \leq \ell'$ to $C'$ of $D'$, for every label $\ell$ which occurs in $I^*$ but not in $C^*$. Let $C'$ be the resulting constraint set:

$$C' = C^* \cup \{\ell \leq \ell \mid \ell \in fl(I^*) \setminus fl(C^*)\}$$

Finally, set $I' = I^*$. Step II. We now argue that the resulting derivation $D_N$ of the judgement

$$t; I'; C'; A \vdash_{CFL} e : \sigma$$

is indeed normal. To do this, let $\text{instcon}(D_N, i) = \langle C', I', A_i, \sigma, \varphi \rangle$ be an arbitrary instantiation context in $\text{Instcon}(D_N)$. We have $A_l = \text{env}(D_N, f)$ and $\sigma = \sigma^e_{C'}$ for some $f$ at instantiation site $f^l$. We must then show the following properties, according to Definition 5:

(i) $\text{dom} \varphi = \text{gen}(A_i, \sigma)$

(ii) $C' \triangleright_{\text{dom} \varphi} \sigma$

(iii) $I' \vdash \sigma \preceq^\ell \angle\ell' \varphi$

(iv) $\forall \ell \in fl(A_i) \cdot I' \vdash \ell \preceq^\ell \angle\ell'$ and $I' \vdash \ell \preceq^\ell \angle\ell$

(v) Whenever $I' \vdash \ell_1 \preceq^\ell \angle\ell_2$ and $I' \vdash \ell_3 \preceq^\ell \angle\ell_4$ with $\ell_1 \neq \ell_2$ and $\ell_3 \neq \ell_4$, then $\ell_2 \neq \ell_3$

(vi) $fl(C') = fl((C')^f)$

with $\sigma = \sigma^e_{C'}$. We consider each property in turn.

(i) $\text{dom} \varphi = \text{gen}(A, \sigma)$ follows from the rules of POLYFLOW_{CFL}.

(ii) $C' \triangleright_{\text{dom} \varphi} \sigma$ follows from the proof transformations on the applications of [Let] and [Rec]. By these transformations, each such rule is applied with a binding premise for $f$ having a binding of the form $A = A_i, j : (\forall \ell' \varphi^j) / f$. By construction of $D_N$, it is the case that $C'$ contains exactly the set of flow constraints $\text{let}(\sigma \preceq \varphi^j)$ where the flow variables in $\ell'$ are all distinct and occur only in $\varphi^j$. It follows directly from this property that $C' \triangleright_{\text{dom} \varphi} \sigma^j$.

(iii) $I' \vdash \sigma \preceq^\ell \angle\ell' \varphi$ follows from the rules of POLYFLOW_{CFL}, for suitable $\sigma'$.

(iv) $\forall \ell \in fl(A_i) \cdot I' \vdash \ell \preceq^\ell \angle\ell'$ and $I' \vdash \ell \preceq^\ell \angle\ell$ follows from the rules of POLYFLOW_{CFL}.

(v) Whenever $I' \vdash \ell_1 \preceq^\ell \angle\ell_2$ and $I' \vdash \ell_3 \preceq^\ell \angle\ell_4$ with $\ell_1 \neq \ell_2$ and $\ell_3 \neq \ell_4$, then $\ell_2 \neq \ell_3$. This property follows from (A.32), as follows. If $I' \vdash \ell_1 \preceq^\ell \angle\ell_2$, then $\ell_1$ is in the domain of $\varphi$, provided $\ell_1 \neq \ell_2$. Therefore, $\ell_2$ is in the range of $\varphi$. Similarly, if $\ell_3 \neq \ell_4$ then $\ell_3$ is in the domain of some substitution $\varphi'$ used at instantiation site $j$. Hence, if $\ell_2 = \ell_3$, then the label $\ell_2$ is in the domain of $\varphi'$. But $\ell_2$ is also in the range of $\varphi$. This contradicts property (A.32) of the construction of $D_N$ and is therefore impossible.
(vi) $fl(C') = fl((C')')$ holds by construction of $C'$ from $C^*$ above. CFL closure cannot introduce any variables not already present in the original constraint set or $I'$. Since $fl(I') \in fl(C')$, the property follows.

**Step III.** We now show that the property

$$I'; C \vdash \ell \rightsquigarrow \ell' \Rightarrow I; C \vdash \ell \rightsquigarrow \ell'$$

holds. But this property is clear from the construction of $D_N$: the set $C'$ was constructed from $C$ by addition of inequalities of the form $\sigma \leq \sigma'$, where $\sigma'$ arises from $\sigma$ by renaming variables, using flow variables that occur nowhere else (in particular, none of these flow labels occur in the term $e$). It follows that the constraint set $C$ can be obtained from the set $C'$ by identifying flow variables. This operation can only increase the flow among the labels in $e$ that can be deduced from $C$ in comparison to $C'$. □

### A.6 Soundness Theorem

**Theorem A.6 (Soundness)** For every judgement

$$t; I; C; A \vdash_{CFL} e : \sigma$$

derivable in $\text{POLYFLOW}_{CFL}$ there exists a judgement

$$C_0; I_0; C_0; A_0 \vdash_{cp} e : \sigma$$

derivable in $\text{POLYFLOW}_{copy}$ such that, for all labels $\ell$ and $\ell'$ occurring in $e$ one has

$$I_0; C_0 \vdash_{cp} \ell \rightsquigarrow \ell' \Rightarrow I; C \vdash_{CFL} \ell \rightsquigarrow \ell'$$

**Proof:** By Lemma A.5, there is a judgement

$$t; I'; C'; A \vdash_{CFL} e : \sigma$$

derivable by a normal derivation $D$ in $\text{POLYFLOW}_{CFL}$ and such that, for $\ell, \ell' \in fl(e)$ one has

$$I'; C' \vdash \ell \rightsquigarrow \ell' \Rightarrow I; C \vdash \ell \rightsquigarrow \ell'$$ (A.33)

By Lemma A.4, the judgement

$$(C')'; (I')^D; (C')'; A^D \vdash_{cp} e : \sigma$$

is derivable in $\text{POLYFLOW}_{copy}$. It is easy to see, from the definition of $(I')^D$, that $(I')^D$ and $I'$ agree on the labels occurring in $e$. Then one has

$$(I')^D; (C')'; A^D \vdash_{cp} \ell \rightsquigarrow \ell' \Rightarrow I'; C' \vdash_{CFL} \ell \rightsquigarrow \ell'$$ (A.34)

Using (A.33) and (A.34) together, we get

$$(I')^D; (C')'; A^D \vdash_{cp} \ell \rightsquigarrow \ell' \Rightarrow I; C \vdash_{CFL} \ell \rightsquigarrow \ell'$$

Taking $C_0 = (C')'$, $I_0 = (I')^D$, $A_0 = A^D$ then establishes the theorem. □