Type-Based Flow Analysis: From Polymorphic Subtyping to CFL-Reachability.

Jakob Rehof Manuel Fähndrich

Microsoft Research
One Microsoft Way, Redmond WA 98052
{rehof,maf}@microsoft.com

Abstract

We present a novel approach to scalable implementation of type-based flow analysis with polymorphic subtyping. Using a new presentation of polymorphic subtyping with instantiation constraints, we are able to apply context-free language (CFL) reachability techniques to type-based flow analysis. We develop a CFL-based algorithm for computing flow information in time $O(n^3)$, where n is the size of the typed program. The algorithm substantially improves upon the best previously known algorithm for flow analysis based on polymorphic subtyping with complexity $O(n^8)$. Our technique also yields the first demand-driven algorithm for polymorphic subtype-based flow-computation. It works directly on higher-order programs with structured data of finite type (unbounded data structures are incorporated via finite approximations), supports context-sensitive, global flow summarization and includes polymorphic recursion.

1 Introduction

Type-based program analyses have received much attention (see, *e.g.*, [Hei95, PO95, Mos96, HM97, NNH99, FRD00b]). Attractive properties of such analyses include: they typically work directly on higher-order programs with structured datatypes, they provide a natural separation between the specification (type system) and implementation of the analysis, and standard techniques from type theory are applicable to reason about properties (*e.g.*, soundness, completeness) of the analysis.

Type-based flow analysis tracks the flow of values by annotating type structure with flow labels ℓ, representing values at specific program points (see, *e.g.*, [Mos96, NNH99]). Answering queries of the form "Does any value at program point ℓ_1 flow to program point ℓ_2" solves many static analysis problems such as finding potential pointer aliases, determining possible targets of indirect function calls, and delimiting storage escapement. This paper studies efficient techniques to answer such queries in the setting of type-based flow analysis with polymorphic subtyping. Based on the polymorphic type structure of the program, our analysis is *context-sensitive*, i.e., it avoids spurious flow between different calling contexts. Subtyping further enhances analysis precision by modeling a *directional* (non-symmetric) notion of value flow, see, *e.g.*, [Hei95].

While each of the features – polymorphism and subtyping – are established as practical components of type inference systems, their *simultaneous combination* in polymorphic subtyping is not: scaling up polymorphic subtype inference to even moderately realistic program sizes is an outstanding open problem. This paper presents a new attack on the scaling problem for subtyping-based polymorphic flow analysis.

The main contributions of this paper are:

- A novel presentation of polymorphic subtyping, using *instantiation constraints* (also known as semi-unification constraints [Hen93]). Based on this presentation, we are able to apply *Context-Free Language (CFL) Reachability* [Yan90, RHS95, MR00] techniques to compute directional, context-sensitive flow information for higher-order programs in polymorphic subtyping systems (including polymorphic recursion).
- Our resulting algorithm improves the asymptotic complexity of the best previously known algorithm [Mos96] based on polymorphic subtyping from $O(n^8)$ to $O(n^3)$, where n is the size of the typed program Programs are explixitly typed, and n measures the size of the program and the size of the explicit types in the program. In theory, n can be exponential in program size but is close to program size in practice [Mit96]).
- Our results open the door to new implementation techniques for flow computation with polymorphic subtyping. First, 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. Second, our algorithm leads to demand-driven techniques, which, to our knowledge, have not been obtained before with polymorphic subtyping.

In a previous paper [FRD00b] we have presented a flow analysis in polymorphic type systems based on instantiation constraints but without subtyping. The present paper provides a substantial generalization of [FRD00b] through the incorporation of subtyping.

In order to keep the present paper within limits, some details and proofs are omitted. These can be found in [FRD00a].

In the remainder of this introduction we describe the language framework we use for our flow analysis.

$\tau \in$ Types

$$\tau \quad ::= \quad int \mid \tau \to \tau \mid \tau \times \tau$$

$e \in$ Terms

$$\begin{array}{rcl} e & ::= & x \mid n \mid (e_1\texttt{,}\ e_2) \mid \lambda x{:}\tau.e \mid e_1\, e_2 \mid \\ & & \texttt{let}\ f\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \mid f^i \mid \\ & & \texttt{letrec}\ f : \tau\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \mid \\ & & \texttt{if0}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid \pi_j\ e \end{array}$$

Figure 1: Object language

$\ell \in$ Labels

$\sigma \in$ Labeled Types

$$\sigma \quad ::= \quad int^\ell \mid \sigma \to^\ell \sigma \mid \sigma \times^\ell \sigma$$

Figure 2: Annotated types

1.1 Object language

Figure 1 shows the language we use to illustrate our flow analysis. It is a higher-order functional language containing integers (n), pairs, lambda-abstractions, conditionals (testing for 0), recursive functions, and pair selection (π_j). Our language is typed using a standard monomorphic type system. Monomorphic types are ranged over by τ. The language distinguishes between λ-bound variables x and let- or letrec-bound variables f. Our object language is *explicitly typed*, *i.e.*, we assume that the type structure of programs is given in the form of type annotations on λ- and letrec-bound variables, written $x : \tau$ and $f : \tau$, respectively. Uses of let- and letrec-bound variables f^i are annotated with an *instantiation site* i, distinguishing all occurrences of such variables.

1.2 Annotated types

Since we are interested in flow, we will annotate type constructors with *flow labels* ℓ, and we will allow labels on expressions, e^ℓ. *Annotated types* are written σ and are defined in Figure 2. Annotated types will be used as a technical device for performing flow analysis on programs of the object language shown in Figure 1, they do not belong to that language itself. Erasing all labels from an annotated type results in a monomorphic type of the object language. The reader should think of the flow label annotations as being superimposed on the types of the object language, and flow analysis will be specified later using a separate type system for assigning annotated types to programs. To distinguish between annotated types and the types of the object language, we sometimes refer to the latter as *underlying types*, and the type system of the object language is called the *underlying type system*.

The underlying types of let- and letrec-bound variables are given monomorphic types in the underlying type system, but label annotations on types of let- and letrec-bound variables will be treated *polymorphically* by our flow analysis. We will allow *polymorphic recursion* [Myc84, Hen93] over flow labels, as found in [Mos96]. We will also allow our flow analysis to exploit *subtyping* relations over flow labels, written $\ell_1 \leq \ell_2$, to represent the fact that a value identified by label ℓ_1 flows to the program point labeled ℓ_2, in the style of [Hei95, Mos96]. As in the case of polymorphism, subtyping belongs to the flow analysis framework and is not part of the object language.

```
let id = λx:int^ℓ1.x^ℓ2
in
   ((id^i 0^ℓ3)^ℓ4 , (id^j 1^ℓ5)^ℓ6)
end
```

Figure 3: Example program e

Each of the features – polymorphism and subtyping over flow labels – enhance the precision of type-based flow analysis [Hei95, Mos96], and when combined into one system we arrive at flow analysis based on *polymorphic subtyping* as studied in this paper. The polymorphic treatment of flow labels will allow a certain form of *context-sensitivity* (values only flow interprocedurally through well-matched call-return sequences of functions) and subtyping allows us to treat flow directionally (the fact that a value at ℓ_1 flows to ℓ_2 does not imply that a value at ℓ_2 flows to ℓ_1).

The language- and analysis framework described above will be held fixed throughout most of the paper. However, in a later section (Section 6) we will indicate how our techniques can be extended to a language whose *underlying* type system is polymorphic.

The remainder of this paper is organized as follows. Section 2 reviews problems in combining polymorphism and subtyping and sketches our solution. Section 3 presents polymorphic subtyping with instantiation constraints. Section 4 defines our flow relation based on CFL-reachability. Section 5 presents a cubic time algorithm for computing all flow queries. Section 6 shows that our techniques extend to polymorphic and recursive types in the underlying type structure. Section 7 discusses related work, and Section 8 concludes.

2 Flow analysis with polymorphic subtyping

Scaling up type inference for polymorphism combined with subtyping remains a challenging problem. In this section, we first identify two major problems with current implementation techniques for flow analysis based on polymorphic subtyping (Section 2.1 and Section 2.2). In Section 2.3 and Section 2.4 we then give an intuitive overview of our solution.

We illustrate our techniques with a very simple example program e, as shown in Figure 3. We are interested in tracking the flow of constants 0 and 1 labeled with ℓ_3 and ℓ_5. We will do so by performing type inference over the annotated types in the program (our explanation will be intuitive, and the type system used will be precisely defined later).

2.1 Problem 1: Constraint copying

All previous work in polymorphic subtype inference is based on *qualified polymorphic types*. In our setting of flow analysis, a qualified polymorphic type has the form

$$\forall \vec{\ell}.C \Rightarrow \sigma$$

In this type, quantification occurs over flow labels, and σ is an annotated type, which typically contains labels that are

quantified. The component C is a set of captured *subtyping constraints* over flow labels of the form $\ell \leq \ell'$, qualifying the type σ. Since C may contain quantified labels from $\vec{\ell}$, such an approach gives rise to copies of the captured constraints at all instantiation sites for that type, as we will illustrate next.

A standard [1] polymorphic constrained type (see, *e.g.*, [Smi94, TS96, Mos96]) for our example function `id` of Figure 3 is

$$\forall \ell_1 \ell_2 . \{\ell_1 \leq \ell_2\} \Rightarrow int^{\ell_1} \rightarrow int^{\ell_2}$$

In this typing, the constraint set $\{\ell_1 \leq \ell_2\}$ captures the fact that any value (represented by ℓ_1) passed as argument to `id` flows to the result of the function (represented by ℓ_2).

In a *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 $\mathtt{id}^i$ and $\mathtt{id}^j$. At instantiation site i, the label ℓ_1 on the domain type of `id` gets copied to the label ℓ_3, because ℓ_3 labels the actual argument `0` at call site i. The label ℓ_2 of the range type of `id` gets copied to the label ℓ_4, because ℓ_4 labels the result of the call. Because the type of `id` is constrained by $\ell_1 \leq \ell_2$, the constraint set as a whole gets copied into the set $\{\ell_3 \leq \ell_4\}$ at site i. By a similar argument, the constraint set $\{\ell_1 \leq \ell_2\}$ gets copied into $\{\ell_5 \leq \ell_6\}$ at site j. After these steps we arrive at the standard polymorphic typing judgment for e

$$\{\ell_3 \leq \ell_4, \ell_5 \leq \ell_6\}; \emptyset \vdash e : int^{\ell_4} \times int^{\ell_6} \qquad (1)$$

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 can read off interesting flow relations. For example, because $\ell_3 \leq \ell_5$ is a constraint in the type (1), we can conclude that the value `0` (ℓ_3) flows to the first component of the resulting pair (ℓ_4). Similarly, we can see that the value `1` (ℓ_5) flows to the second (ℓ_6), as indicated by the constraint $\ell_5 \leq \ell_6$ in (1).

Polymorphism over labels, here implemented via constraint copying, keeps the two instantiation sites apart, matching up a call site (*e.g.*, $\mathtt{id}^i\ \mathtt{0}^{\ell_3}$) with its proper return (ℓ_4). This was achieved by making two distinct copies, during type inference, of the constraint set $\{\ell_1 \leq \ell_2\}$, one copy at instantiation site i, and another at site j. A monomorphic analysis, in contrast, typically predicts, imprecisely, that either value (ℓ_3 or ℓ_5) flows to either return point (ℓ_4 or ℓ_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] In particular, the problem has generated a significant amount of research on *constraint simplification*, which aims at compacting constraint sets before they are copied [FM89, Cur90, Kae92, Smi94, EST95, Mos96, 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].

2.2 Problem 2: Demand-driven flow computation

Constraint copying methods for polymorphic subtyping systems are not demand-driven [Mos96]. For instance, if we only ask for the flow between ℓ_3 and ℓ_4 in our example program `id`, traditional methods still copy the constraint set $\{\ell_1 \leq \ell_2\}$ into both call sites $\mathtt{id}^i$ and $\mathtt{id}^j$, even though only the former copy is necessary to answer the question.

A related, more subtle problem is that flow queries may originate at arbitrary program points. For example, we may ask which values globally flow into the formal parameter `x` of the function definition `id` (in our example, the answer is ℓ_3 and ℓ_5, since `0` is an actual parameter to `id` at call site i, and `1` is an actual parameter to `id` at site j). Because copying does not keep track of the source of constraint copies, it *does not represent the flow of values between a polymorphic function and its instantiations*. Recovering this information in the traditional framework has proven to be nontrivial [Mos96, FFA00], both in terms of computational expense and correctness, because it requires the entire typing derivation rather than merely the final typing judgement of the program. Furthermore, it is unclear how such techniques can naturally accommodate demand-driven versions.

2.3 A new method based on instantiation constraints

Both of the problems mentioned in Section 2.1 and 2.2 inhibit scalability of polymorphic subtype-based flow analysis. We tackle these problems by introducing a new presentation of polymorphic subtyping, based on *instantiation constraints*. Instantiation constraints were used in [FRD00b] to support scalable context-sensitive flow analysis. Here, we generalize the approach by incorporating subtyping, and, as we will see, this will significantly change the way flow computation is done. We refer to our system as $\mathsf{POLYFLOW}_{\mathsf{CFL}}$.

Instead of constrained types $\forall \vec{\ell}.C \Rightarrow \sigma$, $\mathsf{POLYFLOW}_{\mathsf{CFL}}$ uses standard quantified types of the form $\forall \vec{\ell}.\sigma$, which are given meaning in combination with a global set of constraints. In $\mathsf{POLYFLOW}_{\mathsf{CFL}}$, the term e from the example shown in Figure 3 receives a typing of the form

$$\left\{ \begin{array}{l} \ell_1 \preceq^i_{\div} \ell_3, \ell_2 \preceq^i_{+} \ell_4, \\ \ell_1 \preceq^j_{\div} \ell_5, \ell_2 \preceq^j_{+} \ell_6 \end{array} \right\}; \{\ell_1 \leq \ell_2\}; \emptyset \vdash e : \sigma \qquad (2)$$

where $\sigma = int^{\ell_4} \times int^{\ell_6}$. A judgment in $\mathsf{POLYFLOW}_{\mathsf{CFL}}$ has five components, from left to right: a set of instantiation constraints I, a set of flow constraints C, a type environment (here empty), a term and a type.

The instantiation constraints I explicitly represent the label substitutions φ_i and φ_j given by

$$\varphi_i = \{\ell_1 \mapsto \ell_3, \ell_2 \mapsto \ell_4\}$$

and

$$\varphi_j = \{\ell_1 \mapsto \ell_5, \ell_2 \mapsto \ell_6\}$$

These substitutions are used at instantiation sites i and j, respectively, to produce instances of `id`'s type $int^{\ell_1} \rightarrow int^{\ell_2}$. In the constraint set I, the substitution φ_i is represented by the constraints $\ell_1 \preceq^i_{\div} \ell_3, \ell_2 \preceq^i_{+} \ell_4$, and φ_j is represented by $\ell_1 \preceq^j_{\div} \ell_5, \ell_2 \preceq^j_{+} \ell_6$. In a constraint copying framework, φ_i and φ_j are applied at sites i and j to copy the subtype constraint set $\{\ell_1 \leq \ell_2\}$ associated with `id`, yielding $\{\ell_3 \leq \ell_4\}$ at site i and $\{\ell_5 \leq \ell_6\}$ at site j. In our instantiation constraints, The *indices* i and j serve to keep substitutions at distinct instantiation sites apart.

The crucial difference of $\mathsf{POLYFLOW}_{\mathsf{CFL}}$ to copy-based systems is that we avoid constraint copying altogether:

[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.

- *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.*

Having addressed the copy-problem from Section 2.1, we now show how 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 ℓ_3 flows. Drawing flow constraint $\{\ell_1 \leq \ell_2\}$ as a directed edge from ℓ_1 to ℓ_2 and drawing instantiation constraints $\ell_1 \preceq^i_\div \ell_3, \ell_2 \preceq^i_+ \ell_4$ for site i as dotted edges, we recover the flow from ℓ_3 to ℓ_4 at instantiation site i by completing the following diagram, the lower dashed edge representing the "recovered" flow constraint:

$$\begin{tikzcd} \ell_1 \arrow[r, bend left] \arrow[d, dotted, "\preceq^i_\div"] & \ell_2 \arrow[d, dotted, "\preceq^i_+"] \\ \ell_3 \arrow[r, dashed] & \ell_4 \end{tikzcd}$$

Label ℓ_3 represents the actual argument at call-site i and ℓ_4 represents the result at call-site i. Intuitively, there is flow from ℓ_3 to ℓ_4 because the argument flows into the identity function `id` to the formal ℓ_1, within the identity function to the return value ℓ_2, and back out of the identity function to the result ℓ_4 at call site i. The flow is *valid* because the *in-flow* $\div$ and *out-flow* $+$ agree on the instantiation site i.

Note how the negative ($\div$) instantiation edge $\ell_1 \preceq^i_\div \ell_3$ represents the flow of an actual to a formal, and the positive ($+$) edge $\ell_2 \preceq^i_+ \ell_4$ represents the flow of the result to the call-site i. The polarities $p \in \{\div, +\}$ on instantiation edges indicate their flow direction (in-flow or out-flow). Negative $\div$ instantiation edges represent flow in the direction opposite the instantiation, whereas positive $+$ instantiation edges represent flow in the direction of the instantiation.

Polarities are assigned to instantiation constraints according to the polarity of the "source types" of instantiations. In our example, ℓ_1 occurs negatively, ℓ_2 positively in the type $int^{\ell_1} \to int^{\ell_2}$ of `id`. Instantiation polarities were introduced in [FRD00b]. Disregarding polarities and interpreting instantiation constraints as bi-directional flow constraints results in a complete loss of context-sensitivity.

We have sketched how to recover the flow on-demand, using only parts of the constraint systems needed for this query. This addresses the on-demand problem from Section 2.2.

A further advantage of instantiation constraints, addressing the second problem in Section 2.2, is that *all* flow is present in the constraint sets I and C obtained in a typing judgment for $\mathsf{POLYFLOW}_{\mathsf{CFL}}$. Consider for example the flow of both value 0 (ℓ_3) and 1 (ℓ_5) to formal parameter `x` (ℓ_1). We can recover such flow similarly to the flow above via $\ell_1 \preceq^i_\div \ell_3$ and $\ell_1 \preceq^j_\div \ell_5$. As mentioned in Section 2.2, to recover such flow in copy-based systems, the entire typing derivation is required instead of merely the final judgment.

2.4 CFL-reachability

How can we systematically recover global flow from instantiation and flow constraints? As we will show in this paper, this is best formulated as *Context-Free Language (CFL) reachability* on a graph formed by flow and instantiation constraints (edges) and labels (nodes). Let us give the intuition behind this idea. We assume that in this graph negative instantiation constraints are reversed and labeled with opening parentheses $(_i$. Positive instantiation edges are labeled closing parentheses $)_i$, and flow edges with d. All flow paths in the graph now spell words. For example, the path

$$\boxed{C \vdash L \leq L}$$

$$\frac{}{C, \ell_1 \leq \ell_2 \vdash \ell_1 \leq \ell_2}[\text{Id}]$$

$$\frac{}{C \vdash \ell \leq \ell}[\text{Refl}]$$

$$\frac{C \vdash \ell_0 \leq \ell_1 \qquad C \vdash \ell_1 \leq \ell_2}{C \vdash \ell_0 \leq \ell_2}[\text{Trans}]$$

Figure 4: Constraint relation

$$\ell_3 \xrightarrow{(_i} \ell_1 \xrightarrow{\mathsf{d}} \ell_2 \xrightarrow{)_i} \ell_4$$

spells the word "$(_i\mathsf{d})_i$". The invalid flow path

$$\ell_3 \xrightarrow{(_i} \ell_1 \xrightarrow{\mathsf{d}} \ell_2 \xrightarrow{)_j} \ell_6$$

spells the word "$(_i\mathsf{d})_j$" and corresponds to calling `id` at instance i, but returning to instance j. Valid and invalid flow are distinguished by membership in a particular language. For our example, the language contains the words $(_i\mathsf{d})_i$ and $(_j\mathsf{d})_j$, but no others. In general, valid flow paths are characterized by words with matching sets of parentheses. Spurious flow paths are simply not part of that language.

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. In contrast to this work, our analysis allows context-sensitive tracking of the flow of values through any finite data-type, and it directly incorporates finitely typed higher-order functions, since function types are just another kind of data-type in type-based analysis. We note that, in higher-order programs, a function symbol may occur in contexts that are not call sites and matched flow may not correspond to actual calls and returns. Well-matched paths are also used in [MR00], where an analysis is described for higher-order programs manipulating arbitrary structured data. However, this analysis is context-insensitive, whereas the presence of polymorphism in our type system provides for a form of context-sensitivity. On the other hand, we can only handle unbounded data-structures (such as lists) in an approximative way, as shown in Section 6. The reader is referred to Section 7 for more comparisons with previous work.

3 Polymorphic subtyping with instantiation constraints

$\mathsf{POLYFLOW}_{\mathsf{CFL}}$ uses polymorphic types over labels of the form $\forall\vec{\ell}.\sigma$ without qualifying constraints. Judgments have the form

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

$\boxed{C \vdash \sigma \leq \sigma}$

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

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

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

Figure 5: Subtype relation

$\boxed{I \vdash \sigma \preceq_p^i \sigma \qquad I \vdash \ell \preceq_p^i \ell}$

$$\frac{}{I, \ell \preceq_p^i \ell' \vdash \ell \preceq_p^i \ell'}[\text{Id}]$$

$$\frac{I \vdash \ell \preceq_p^i \ell'}{I \vdash int^{\ell} \preceq_p^i int^{\ell'}}[\text{Int}]$$

$$\frac{I \vdash \ell \preceq_p^i \ell' \qquad I \vdash \sigma_1 \preceq_p^i \sigma_1' \qquad I \vdash \sigma_2 \preceq_p^i \sigma_2'}{I \vdash \sigma_1 \times^{\ell} \sigma_2 \preceq_p^i \sigma_1' \times^{\ell'} \sigma_2'}[\text{Pair}]$$

$$\frac{I \vdash \ell \preceq_p^i \ell' \qquad I \vdash \sigma_1 \preceq_{\overline{p}}^i \sigma_1' \qquad I \vdash \sigma_2 \preceq_p^i \sigma_2'}{I \vdash \sigma_1 \rightarrow^{\ell} \sigma_2 \preceq_p^i \sigma_1' \rightarrow^{\ell'} \sigma_2'}[\text{Fun}]$$

Figure 6: Instantiation Relation

shown in Figure 7. Here, I is a set of *instantiation constraints*, C is a set of *flow constraints* on labels, and A is a *type environment* assigning types to free variables in term e. Such a judgment means that, under the assumptions contained in I, C and A, term e has labeled type σ.

The subtyping rule [Sub] of Figure 7 uses standard logic, shown in Figure 4 and Figure 5, for flow judgments $C \vdash \ell \leq \ell'$ and subtyping judgments $C \vdash \sigma \leq \sigma'$, where C is a set of flow constraints on labels, $\ell \leq \ell'$.

Type environment A contains two kinds of assumptions. One has the form $A', f : (\forall \vec{\ell}.\sigma, \vec{\ell'})$, where f is a `let`- or `letrec`-bound variable. Here, $\forall \vec{\ell}.\sigma$ is the quantified type assumed for f, and $\vec{\ell'}$ is a vector containing the labels that are free in A'. The notation $fl(\sigma)$ and $fl(A)$ denotes the free labels occurring in σ and A, respectively. The remaining type assumptions have the form $A', x : \sigma$ assigning (non-quantified) labeled types to λ-bound variables.

Rule [Let] binds a `let`-bound variable f to a quantified type, and rule [Rec] binds a `letrec`-bound variable f to a quantified type. Notice that, in rule [Rec], the quantified type is used in typing the term e_1 in the `letrec`-binding, thereby allowing polymorphic recursion over labels. In rule [Lam] and [Rec] we use the notation $|\sigma|$ to denote the underlying type that arises from the annotated type σ by erasing all label annotations from σ. In rule [Lam] we require $|\sigma| = \tau$, *i.e.*, σ must have the same structure as τ, and similarly in rule [Rec] for σ_1 and τ.

An instantiation constraint $\ell \preceq_p^i \ell'$ states that ℓ instantiates to ℓ' at site i with polarity p. Instantiation constraints represent substitutions at instantiation sites. A polarity p is either positive $+$ or negative $\div$, and $\overline{p}$ negates the polarity of p. Sets of instantiation constraints are written I. Figure 6 lifts instantiations to labeled types, where $I \vdash \sigma \preceq_p^i \sigma'$ expresses that σ instantiates to σ' given I.

Instantiation constraints I must satisfy: for any particular index i and any label ℓ, one has[3]:

$$\forall \ell'.\ \ell \preceq_p^i \ell' \in I \wedge \ell \preceq_{p'}^i \ell'' \in I \Rightarrow \ell' = \ell'' \qquad (3)$$

A constraint set I thus gives rise to a collection of substitutions φ_i, indexed by instantiation site i, where

$$\varphi_i(\ell) = \ell' \text{ if } \ell \preceq_p^i \ell' \in I$$

and the identity everywhere else. We use the notation

$$I \vdash \sigma \preceq_p^i \sigma' : \varphi$$

to mean that given I, σ instantiates to σ' under φ (*i.e.*, $\varphi(\sigma) = \sigma'$).

Instantiation constraints are used in rule [Inst] in Figure 7. From the assumption $A, f : (\forall \vec{\ell}.\sigma, \vec{\ell'})$ we derive the type σ' for f^i at instantiation site i, provided that

$$I \vdash \sigma \preceq_p^i \sigma' : \varphi$$

holds for some φ with $dom\,\varphi = \vec{\ell}$. Moreover, for the labels $\vec{\ell'}$, which are unquantifiable at the point of `let`- or `letrec`-binding, we require [4]

$$I \vdash \vec{\ell'} \preceq_+^i \vec{\ell'} \text{ and } I \vdash \vec{\ell'} \preceq_\div^i \vec{\ell'}$$

which have the effect that for all ℓ' in $\vec{\ell'}$, there are constraints $\ell' \preceq_+^i \ell'$ and $\ell' \preceq_\div^i \ell'$ in I.

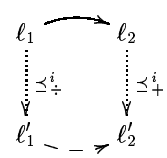
In rule [Label], $\sigma[\ell]$ denotes the type σ with top level label ℓ. The rule allows a subterm e to be labeled with the top-level label of its type, so that flow queries on labels can refer to arbitrary subterms in the program. The remaining rules of Figure 7 are standard type rules.

4 Flow relation

This section defines our flow relation and flow-graphs induced by typing derivations. The presence of instantiation constraints in the subtyping system leads to a clean flow logic, which is implementable via CFL-reachability.

4.1 Flow logic

Given a derivation of $I; C \vdash_{\mathsf{CFL}} e : \sigma$, 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. As was explained in Section 1, we must recover flow from ℓ_1' to ℓ_2' in the following situation:



[3] The reader familiar with semi-unification will note that this rule is enforced by the algorithm of [Hen93].

[4] For $\vec{\ell'} = \ell_1', \ldots, \ell_n'$ we write $I \vdash \vec{\ell'} \preceq_p^i \vec{\ell'}$ as shorthand for $I \vdash \ell_j' \preceq_p^i \ell_j'$, $j = 1 \ldots n$.

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

Base Rules

$$\frac{}{I; C; A, x : \sigma \vdash_{\mathsf{CFL}} x : \sigma}[\text{Id}] \qquad \frac{}{I; C; A \vdash_{\mathsf{CFL}} n^{\ell} : int^{\ell}}[\text{Int}]$$

$$\frac{I; C; A \vdash_{\mathsf{CFL}} e_1 : \sigma_2 \to^{\ell} \sigma_1 \qquad I; C; A \vdash_{\mathsf{CFL}} e_2 : \sigma_2}{I; C; A \vdash_{\mathsf{CFL}} e_1\ e_2 : \sigma_1}[\text{App}]$$

$$\frac{I; C; A, x : \sigma \vdash_{\mathsf{CFL}} e : \sigma' \qquad |\sigma| = \tau}{I, C; A \vdash_{\mathsf{CFL}} \lambda^{\ell} x{:}\tau.e : \sigma \to^{\ell} \sigma'}[\text{Lam}]$$

$$\frac{I; C; A \vdash_{\mathsf{CFL}} e_1 : \sigma_1 \qquad I; C; A \vdash_{\mathsf{CFL}} e_2 : \sigma_2}{I; C; A \vdash_{\mathsf{CFL}} (e_1, e_2)^{\ell} : \sigma_1 \times^{\ell} \sigma_2}[\text{Pair}]$$

$$\frac{I; C; A \vdash_{\mathsf{CFL}} e : \sigma_1 \times^{\ell} \sigma_2}{I; C; A \vdash_{\mathsf{CFL}} \pi_j\ e : \sigma_j}[\text{Proj } j = 1, 2]$$

$$\frac{I; C; A \vdash_{\mathsf{CFL}} e_0 : int^{\ell} \qquad I; C; A \vdash_{\mathsf{CFL}} e_1 : \sigma \qquad I; C; A \vdash_{\mathsf{CFL}} e_2 : \sigma}{I; C; A \vdash_{\mathsf{CFL}} \texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \sigma}[\text{Cond}]$$

$$\frac{I; C; A \vdash_{\mathsf{CFL}} e : \sigma \qquad C \vdash \sigma \leq \sigma'}{I; C; A \vdash_{\mathsf{CFL}} e : \sigma'}[\text{Sub}] \qquad \frac{I; C; A \vdash_{\mathsf{CFL}} e : \sigma[\ell]}{I; C; A \vdash_{\mathsf{CFL}} e^{\ell} : \sigma}[\text{Label}]$$

Polymorphic Rules

$$\frac{\begin{array}{l} I; C; A \vdash_{\mathsf{CFL}} e_1 : \sigma_1 \\ I; C; A, f : (\forall \vec{\ell}.\sigma_1, \vec{\ell'}) \vdash_{\mathsf{CFL}} e_2 : \sigma_2 \\ \vec{\ell} = gen(A, \sigma_1) \qquad \vec{\ell'} = fl(A) \end{array}}{I; C; A \vdash_{\mathsf{CFL}} \texttt{let}\ f\ \texttt{=}\ e_1\ \texttt{in}\ e_2 : \sigma_2}[\text{Let}]$$

$$\frac{\begin{array}{l} I; C; A, f : (\forall \vec{\ell}.\sigma_1, \vec{\ell'}) \vdash_{\mathsf{CFL}} e_1 : \sigma_1 \\ I; C; A, f : (\forall \vec{\ell}.\sigma_1, \vec{\ell'}) \vdash_{\mathsf{CFL}} e_2 : \sigma_2 \\ \vec{\ell} = gen(A, \sigma_1) \qquad \vec{\ell'} = fl(A) \qquad |\sigma_1| = \tau \end{array}}{I; C; A \vdash_{\mathsf{CFL}} \texttt{letrec}\ f{:}\tau\ \texttt{=}\ e_1\ \texttt{in}\ e_2 : \sigma_2}[\text{Rec}]$$

$$\frac{I \vdash \sigma \preceq^{i}_{+} \sigma' : \varphi \qquad dom\,\varphi = \vec{\ell} \qquad I \vdash \vec{\ell'} \preceq^{i}_{+} \vec{\ell'} \qquad I \vdash \vec{\ell'} \preceq^{i}_{\div} \vec{\ell'}}{I; C; A, f : (\forall \vec{\ell}.\sigma, \vec{\ell'}) \vdash_{\mathsf{CFL}} f^{i} : \sigma'}[\text{Inst}]$$

$gen(A, \sigma) = fl(\sigma) \setminus fl(A)$

Figure 7: CFL-Based System $\mathsf{POLYFLOW}_{\mathsf{CFL}}$

$\boxed{I; C \vdash_{\mathsf{CFL}} \ell \rightsquigarrow \ell}$

$$\frac{C \vdash \ell_1 \leq \ell_2}{I; C \vdash \ell_1 \rightsquigarrow_p \ell_2}[\text{Level}]$$

$$\frac{}{I, \ell_1 \preceq^{i}_{+} \ell_2; C \vdash \ell_1 \rightsquigarrow_{+} \ell_2}[\text{Out}]$$

$$\frac{}{I, \ell_1 \preceq^{i}_{\div} \ell_2; C \vdash \ell_2 \rightsquigarrow_{\div} \ell_1}[\text{In}]$$

$$\frac{I; C \vdash \ell_0 \rightsquigarrow_p \ell_1 \qquad I; C \vdash \ell_1 \rightsquigarrow_p \ell_2}{I; C \vdash \ell_0 \rightsquigarrow_p \ell_2}[\text{Trans}]$$

$$\frac{I \vdash \ell_1 \preceq^{i}_{\div} \ell_0 \qquad I; C \vdash \ell_1 \rightsquigarrow_{\mathsf{m}} \ell_2 \qquad I \vdash \ell_2 \preceq^{i}_{+} \ell_3}{I; C \vdash \ell_0 \rightsquigarrow_p \ell_3}[\text{Match}]$$

$$\frac{I; C \vdash \ell_0 \rightsquigarrow_{+} \ell_1 \qquad I; C \vdash \ell_1 \rightsquigarrow_{\div} \ell_2}{I; C \vdash_{\mathsf{CFL}} \ell_0 \rightsquigarrow \ell_2}[\text{Stage}]$$

$p = +, \div, \mathsf{m}$

Figure 8: Flow relation for $\mathsf{POLYFLOW}_{\mathsf{CFL}}$

Our solution is given in Figure 8 axiomatizing judgments $I; C \vdash_{\mathsf{CFL}} \ell_1 \rightsquigarrow \ell_2$. This judgment means that, under constraints I and C, there is valid flow from label ℓ_1 to ℓ_2. Some flow paths only involve flow constraints in C and are deduced directly from C via rule [Level], i.e., if $C \vdash \ell_1 \leq \ell_2$ is derivable, then we deduce flow from ℓ_1 to ℓ_2. Internally, the rules use auxiliary judgments of the form $I; C \vdash \ell_1 \rightsquigarrow_+ \ell_2$, $I; C \vdash \ell_1 \rightsquigarrow_\div \ell_2$ and $I; C \vdash \ell_1 \rightsquigarrow_{\mathsf{m}} \ell_2$, which define *positive*, *negative* and *matched flow*, respectively. Rules with an occurrence of p are actually rule schemes for all rules obtained by selecting $+, \div$ or m for p. Rule [Trans] encodes transitivity for all auxiliary relations.

Rule [Match] introduces matched flow, of the form $I; C \vdash \ell_1 \rightsquigarrow_{\mathsf{m}} \ell_2$, capturing that the sequence of instantiation edges traversed along a matched flow path forms a well-parenthesized string, where $\succeq^i_\div$ matches $\preceq^i_+$. Matched flow allows us to recover, on demand, the flow represented by copies of constraint systems in constraint copying frameworks. 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.

The remaining rules [In], [Out], and [Stage] encode that some flow paths may contain unmatched initial closing parentheses and unmatched final open parentheses. The unmatched parentheses arise because a flow path may start and end in scopes other than the top-level scope of the program.

4.2 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_{CFL} with nodes L, and the following labeled edges:

$$\begin{array}{l} \ell_1 \xrightarrow{\mathsf{p}} \ell_2 \\ \ell_1 \xrightarrow{(_i} \ell_2 \end{array} \text{ if } \ell_1 \preceq^i_+ \ell_2 \in I$$

$$\begin{array}{l} \ell_2 \xrightarrow{\mathsf{n}} \ell_1 \\ \ell_2 \xrightarrow{)_i} \ell_1 \end{array} \text{ if } \ell_1 \preceq^i_\div \ell_2 \in I$$

$$\ell_1 \xrightarrow{\mathsf{d}} \ell_2 \text{ if } \ell_1 \leq \ell_2 \in C$$

Edges with labels p (n) correspond to positive (negative) instantiation edges used in the [Out] ([In]) rule of the flow relation. Edges with labels $(_i$ and $)_i$ correspond to the instantiation edges used in the [Match] rule, and d-edges represent flow constraints.

A flow relation $I; C \vdash_{\mathsf{CFL}} \ell_1 \rightsquigarrow \ell_2$ is derivable via rules of Figure 8 if and only if there exists a path in G_{CFL} where the sequence of labels along the path from ℓ_1 to ℓ_2 is accepted by the following grammar with start symbol S:

$$\begin{array}{rcl} S & \leftarrow & P\ N \\ P & \leftarrow & M\ P \\ & | & \mathsf{p}\ P \\ & | & \epsilon \\ N & \leftarrow & M\ N \\ & | & \mathsf{n}\ N \\ & | & \epsilon \\ M & \leftarrow & (_i\ M\ )_i \\ & | & M\ M \\ & | & \mathsf{d} \\ & | & \epsilon \end{array}$$

Productions for P accept paths that corresponds to positive flow $I; C \vdash \ell_1 \rightsquigarrow_+ \ell_2$, productions for N produce negative flow paths, and productions for M produce matched flow paths.

In practice, the graph G_{CFL} need not be computed explicitly. Instead, the graph closure can be computed directly on the set of constraints C and I.

4.3 Examples

Figure 9 shows the flow graph and the CFL-graph for an example program. Function `idpair` is the identity on integer pairs. It is instantiated at site i within `f`, which in turn is instantiated at site j.

The top contains the flow graph G with flow and instantiation edges. The bottom contains the corresponding CFL graph G_{CFL} where we omit the d labels on flow edges. There is a flow path from b to z:

$$\ell_b \xrightarrow{(_i} \bullet \xrightarrow{\mathsf{d}} \bullet \xrightarrow{)_i} \ell_1 \xrightarrow{\mathsf{p}} \ell_z$$

where the flow from ℓ_b to ℓ_1 forms a matched flow path that is explicit in a system based on constraint copying.

One advantage of using the type-based approach to flow analysis presented here is that it deals directly with higher-order programs. The type-theoretic notions of *polarity* and *contra-variance* are essential here, and are exploited in our annotation of polarities on instantiation edges. Appendix A contains an example (Figure 13) of a higher-order flow graph and its flow interpretation.

Interestingly, the flow relation of [FRD00b] without subtyping emerges by collapsing all M-productions to ϵ, corresponding to an equational interpretation of the subtype relation. The flow relation thereby becomes *regular*, and flow queries become linear time in the size of the flow graph. Our present results show that the addition of subtyping corresponds to a passage from regular to context-free flow.

4.4 Soundness

Soundness of our flow relation $I; C \vdash_{\mathsf{CFL}} \ell_1 \rightsquigarrow \ell_2$ is non-obvious and requires proof, which we provide in our technical report [FRD00a]. To prove soundness we use a labeled polymorphic subtyping system called $\mathsf{POLYFLOW}_{\mathsf{copy}}$ (we refer the reader to [FRD00a] for the definition). This system is similar to a standard copy-based system studied by Mossin in [Mos96]. It induces a flow relation, denoted $\vdash_{cp}$, which arises from the relation $\vdash_{\mathsf{CFL}}$ (Figure 8) by removing rule [Match]. The soundness theorem states that our flow relation $\vdash_{\mathsf{CFL}}$ is a sound approximation to the relation $\vdash_{cp}$. Since soundness has been established for copy-based systems by Mossin [Mos96], soundness of our notion of flow follows. Technically, we show

Theorem 4.1 *(Soundness) For every judgment*

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

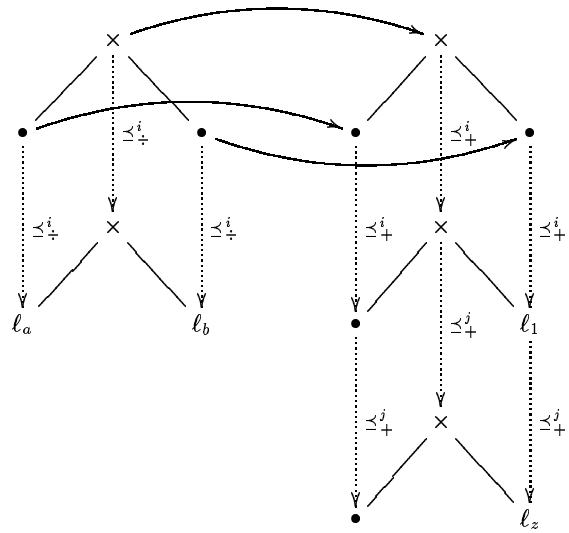
derivable in $\mathsf{POLYFLOW}_{\mathsf{CFL}}$ *there exist* C_0, I_0, A_0 *such that the judgment*

$$C_0; I_0; C_0; A_0 \vdash e : \sigma$$

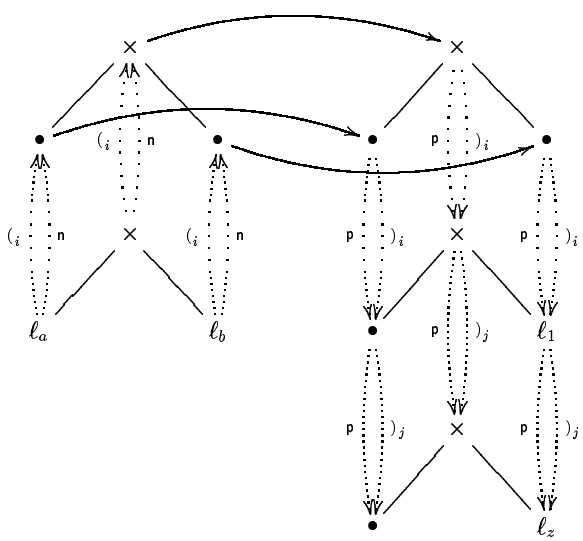
is derivable in $\mathsf{POLYFLOW}_{\mathsf{copy}}$ *and such that, for all labels* ℓ *and* ℓ' *occurring in* e *one has*

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

```
let idpair = λx:int × int.x in
let f      = λy:int.idpair^i (a^ℓa,b^ℓb) in
let z      = (π2 (f^j 0))^ℓz
...
```



Flow graph G



CFL Graph G_{CFL}

Figure 9: $\mathsf{POLYFLOW}_{\mathsf{CFL}}$ example

$$
\begin{array}{rcl}
S & \leftarrow & P\ N \\
P & \leftarrow & M\ P \\
 & | & \mathsf{p}\ P \\
 & | & \epsilon \\
N & \leftarrow & M\ N \\
 & | & \mathsf{n}\ N \\
 & | & \epsilon \\
K_i & \leftarrow & M\ )_i \\
M & \leftarrow & (_i\ K_i \\
 & | & M\ M \\
 & | & \mathsf{d} \\
 & | & \epsilon
\end{array}
$$

Figure 10: Grammar for CFL queries.

This theorem implies that, for any program typable in $\mathsf{POLYFLOW}_{\mathsf{CFL}}$, the induced flow logic $\vdash_{CFL}$ over-approximates the induced relation $\vdash_{cp}$ of $\mathsf{POLYFLOW}_{\mathsf{copy}}$. A core step in the proof consists in showing that our notion of CFL-based flow *recovers* all implicit substitutions φ_i on constraint systems needed in copy-based derivations, as explained in Section 2.3. The presence of polymorphic recursion is a complicating factor for the proof, and our proof uses ideas introduced in [Mos96].

Conversely, by turning label substitutions into instantiation constraints, it is easy (tedious, but not difficult) to see that any typing in $\mathsf{POLYFLOW}_{\mathsf{copy}}$ gives rise to a typing in $\mathsf{POLYFLOW}_{\mathsf{CFL}}$ such that the induced flow relation of $\mathsf{POLYFLOW}_{\mathsf{CFL}}$ is soundly approximated by the induced flow relation of $\mathsf{POLYFLOW}_{\mathsf{copy}}$.

5 Algorithm

Let e be an explicitly typed program, and let n denote its textual size. Thus, n measures the size of the program text together with the type annotations. Furthermore, let m denote the textual size of the type erasure of e (the type erasure of e arises from e by deleting all type annotations from e). Thus, m measures the size of e when type annotations are disregarded. We show how to compute flow queries for a term e of type-erasure size m, and with type-annotated size n. Our algorithm takes $O(n^3)$ time for computing individual or all queries. In the worst case, the size n is exponentially larger than m, although n will typically be close to m in practice [Mit96].

5.1 Constraint derivation

The type rules in Figure 7 directly serve as constraint inference rules. As is standard for inference systems, the use of the subsumption rule [Sub] is 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 6 and 4 in reverse, i.e., to obtain the conclusion, we generate the constraints required by the antecedents. To guarantee the well-formedness of instantiation constraints, rule (3) must be enforced.

After constraint derivation we produce the graph G_{CFL} according to the description given in Section 4.2.

```
    W = edges of G_CFL
    G_0 = ∅
    for each production A ← ε and node ℓ
      add ℓ -A-> ℓ to W;
    while W not empty
      remove edge e = ℓ_1 -B-> ℓ_2 from W;
(1)   if e not in G_0 do
        add e to G_0;
        for each rule r of the form A ← B
(2)       add ℓ_1 -A-> ℓ_2 to W;

        for each rule r of the form A ← B C
          add ℓ_1 to pred_r(ℓ_2);
          for each ℓ_3 in succ_r(ℓ_2)
(3)         add ℓ_1 -A-> ℓ_3 to W;
          end
        end

        for each rule r of the form A ← C B
          add ℓ_2 to succ_r(ℓ_1);
          for each ℓ_0 in pred_r(ℓ_1)
(4)         add ℓ_0 -A-> ℓ_2 to W;
          end
        end
      endif
    end
```

Figure 11: CFL algorithm

5.2 Cubic time algorithm

We follow [MR97, MR00] 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 10. The generic CFL-algorithm in [MR97, MR00] has worst case complexity $O(|\Sigma|^3 n^3)$, where Σ is the set of terminals and non-terminals used. Our grammar has $|\Sigma| = O(m)$.

As is observed in [MR97, MR00], the generic upper bound of $O(|\Sigma|^3 n^3)$ can be substantially improved in specific applications by exploiting the properties of the grammars that arise. Thus, we can improve the upper bound $O(m^3 n^3)$ by using the generic algorithm of Figure 11 and taking advantage of specific properties of our grammar. We show that the algorithm runs in time $O(n^3)$ for the grammar in Figure 10. The proof of the following theorem is given in Appendix B.

Theorem 5.1 *Given I, C, ℓ, ℓ', answering a flow query $I; C \vdash_{\mathsf{CFL}} \ell \rightsquigarrow \ell'$ is decidable in time $O(n^3)$. Moreover, the entire flow relation derivable from I and C is computable in time $O(n^3)$.*

This result improves the best previously known algorithm, given by Mossin [Mos96], from $O(n^8)$ to $O(n^3)$, when polymorphic recursion is included and from $O(n^7)$ to $O(n^3)$ when it is not. The gain is in both cases 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.

On termination, the algorithm produces a graph G_0 containing all possible edges labeled by non-terminals of the grammar. To answer a query for flow from ℓ_1 to ℓ_2 we simply need to inspect G_0 for an S-edge from ℓ_1 to ℓ_2.

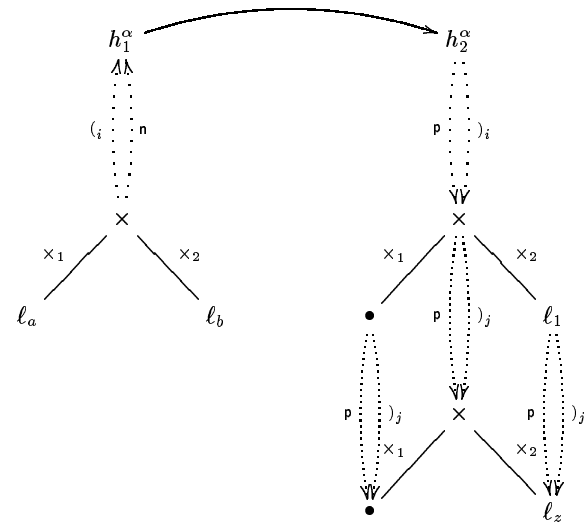


Figure 12: Type polymorphic example

5.3 Demand-driven algorithm

Although the algorithm in Figure 11 is not directly demand-driven, the CFL-formulation allows straight-forward adaptation of the technique in [HRS95, Rep98], yielding a demand-driven algorithm.

6 Polymorphic and recursive type structure

In this section we show that our techniques remain effective, when the underlying types are polymorphic and recursive.

6.1 Polymorphic type structure

$\mathsf{POLYFLOW}_{\mathsf{CFL}}$ type derivations are polymorphic in labels but not in the underlying type structure. Extending polymorphism to the underlying type structure results in more complicated flow queries. Since type variables are instantiated to arbitrary labeled types at instances, flow paths that traverse a type polymorphic function may involve traversing type constructor edges from child to parent and back. Consider the example from Figure 9, but with `idpair` being the polymorphic identity function, instead of the identity on integer pairs. The resulting type instantiation graph is shown in Figure 12. We use h^α for labels annotating type variable α. The flow path from ℓ_b to ℓ_z has the form

$$\ell_b \xrightarrow{[_{\times_2}} \times \xrightarrow{(_i} h_1^\alpha \rightarrow h_2^\alpha \xrightarrow{)_i} \times \xrightarrow{]_{\times_2}} \ell_1 \xrightarrow{\mathsf{p}} \ell_z$$

and involves traversing constructor edges between pair types and their right child. We label constructor edges with the constructor c and the child index j. Traversing such an edge from child to parent corresponds to an opening parenthesis $[_{c_j}$, and traversing from parent to child corresponds to a closing parenthesis $]_{c_j}$. At first sight, the resulting flow queries appear to involve *interleaved matchings* of instantiation parentheses and constructor parentheses. In general,

in the presence of unbounded data structures such as lists, interleaved matchings define the intersection of two context free languages, and flow-relations induced from them become undecidable by a recent result of Reps [Rep00]. In the absence of recursive types however, polymorphic type structure is bounded and guarantees that there always exists a flow path where the two matchings are perfectly nested (this follows abstractly from the fact that the intersection of a context-free language and a regular language is context-free). As a result, the path is given by a single context-free grammar, and all flow queries remain computable in $O(n^3)$ time. Our observations above show that bounded type structure can generally be exploited to eliminate the interleaved matching problem. This may, however, come at the cost of approximating the flow relation when unbounded data structures are present, as we will discuss next.

6.2 Recursive types

Recursive types represent regular infinite type structure. The most general labeling of such structures involves infinitely many distinct labels. The undecidability result of Reps [Rep00] suggests that there exists no partial labeling without loss of precision. However, we can use *regular* labelings of recursive types, where labels repeat in recursive unfoldings. Such labelings introduce spurious flow between unrelated parts of the recursive type but enable us to compute a finite and sound approximation of the interprocedural flow. This technique again amounts to exploiting bounded type structure (which in the case of recursive types is imposed by approximation) to eliminate the interleaved matching problem. Our report [FRD00a] has more details.

7 Related work

Mossin [Mos96] studies the problem of interprocedural flow computation with polymorphic subtyping including polymorphic recursion. His system is based on constraint copying and simplification. Our techniques improve the asymptotic complexity of flow computations and obviates the need to copy and simplify constraints during inference. We prove soundness by reduction to the soundness of the system studied by Mossin. In [Mos98], Mossin describes type-based, higher-order value flow graps. In this work, it is noted that it would be desirable if type-based flow graphs could be used to express well-matched flow paths in the style of [RHS95] (see below). Our work shows that, indeed, this is possible.

Our use of instantiation constraints draws on Henglein's work on semi-unification [Hen93]. Dussart et. al. [DHM95a] present a copy-based algorithm for binding time analysis using subtyping and polymorphic recursion. The unpublished work [DHM95b] seeks a solution in terms of semi-unification. Our work shares motivation with [DHM95b] but our main results on matched flow and CFL-reachability are new.

The work of Reps, Horwitz and Sagiv [RHS95, MR00] provided the connection between CFL-reachability problems and interprocedural, context-sensitive analysis. The work [RHS95] concerns *context-sensitive* (interprocedurally precise) analysis of first order programs manipulating atomic data. Well-matched paths are used to select interprocedurally valid call-return sequences. The work [MR00] contains a higher-order but context-*in*sensitive analysis. Well-matched paths are used for *data-dependence analysis*, *i.e.*, to model cancellation properties of data constructors and destructors to track the flow of data through data-structures. Reps [Rep00] has shown that the combination of the two techniques – context-sensitive data-dependence analysis – results in an uncomputable analysis problem. In contrast, our type-based techniques concern context-sensitive (in the sense of type polymorphism) analysis of higher-order programs manipulating possibly structured data of finite type. Unbounded data-structures can be incorporated via finite approximation using recursive types. The techniques of [HRS95, Rep98] for answering flow queries on demand transfer to our setting via our CFL-reachability formulation.

By incorporating subtyping, the present paper provides a substantial generalization of [FRD00b], where we describe a flow analysis with instantiation constraints but without subtyping. In this setting, CFL-reachability specializes to linear-time graph reachability.

In Lackwit [OJ97], O'Callahan and Jackson define a relation called compatibility. Compatibility is undirected and can be understood as a special case of our flow relation, similar to the one in [FRD00b].

Heintze and McAllester [HM97] present a demand-driven, type-based flow-analysis for ML. Like ours, their analysis traces flow paths on type graphs, but flow paths are not context-sensitive.

Gustavsson and Svenningsson [GS00] have recently developed a usage analysis with polymorphic subtyping using *constraint abstractions* to succinctly represent constraint systems. It is possible that their constraint abstractions could be used to obtain flow analysis with properties similar to ours.

8 Conclusion

We have presented an $O(n^3)$ algorithm for computing context-sensitive, directional flow information for higher-order programs (n is the size of the typed program). This substantially improves on the best previously known $)(n^8)$ algorithm. Our technique is based on a novel presentation of polymorphic subtyping with instantiation constraints. We thereby applied CFL-reachability techniques to the setting of type-based and higher-order flow analysis with structured data of finite type. Unbounded data-structures can be incorporated via finite approximation using recursive types. A novel aspect of our techniques is that they obviate the need to copy constraint systems globally, and they support demand-driven flow computation over a clean, graph-based abstraction. This may turn out to be a major benefit of our approach in practice.

Acknowledgements

We would like to thank David Melski and Tom Reps for valuable discussions concerning context-free language reachability. We would also like to thank Fritz Henglein for discussions on polymorphic recursion. Finally, we would like to thank our reviewers for their detailed comments on this paper.

References

[AWP97] A. Aiken, E.L. Wimmers, and J. Palsberg. Optimal representations of polymorphic types with subtyping. In *Proceedings Theoretical Aspects of Computer Software*, pages 47–77. LNCS 1281, 1997.

[Cur90] P. Curtis. Constrained quantification in polymorphic type analysis. Technical Report CSL-90-1, Xerox Parc, 1990.

[DHM95a] Dirk Dussart, Fritz Henglein, and Christian Mossin. Polymorphic recursion and subtype qualifications: Polymorphic binding-time analysis in polynomial time. In *Proc. 2nd Int'l Static Analysis Symposium (SAS)*, LNCS 983, 1995.

[DHM95b] Dirk Dussart, Fritz Henglein, and Christian Mossin. Polymorphic recursion and subtype qualifications: Polymorphic binding-time analysis in polynomial time. Unpublished draft, May 1995.

[EST95] J. Eifrig, S. Smith, and V. Trifonov. Sound polymorphic type inference for objects. In *Proceedings OOPSLA '95*, 1995.

[FA96] M. Fähndrich and A. Aiken. Making set-constraint program analyses scale. In *Workshop on Set Constraints, Cambridge MA*, 1996.

[FF97] C. Flanagan and M. Felleisen. Componential set-based analysis. In *Programming Language Design and Implementation*, June 1997.

[FFA00] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. Polymorphic versus monomorphic flow-insensitive points-to analysis for c. In *Proceedings of the 7th International Static Analysis Symposium*, June 2000.

[FM89] Y. Fuh and P. Mishra. Polymorphic subtype inference: Closing the theory-practice gap. In *Proc. Int'l J't Conf. on Theory and Practice of Software Development*, pages 167–183, March 1989.

[FRD00a] M. Fähndrich, J. Rehof, and M. Das. From polymorphic subtyping to CFL reachability: Context-sensitive flow analysis using instantiation constraints. Technical Report MSR-TR-99-84, Microsoft Research, 2000. Available at http://research.microsoft.com/~rehof/publications.html.

[FRD00b] M. Fähndrich, J. Rehof, and M. Das. Scalable context-sensitive flow analysis using instantiation constraints. In *Programming Language Design and Implementation*, June 2000.

[GS00] J. Gustavsson and J. Svenningsson. A usage analysis with bounded usage polymorphism and subtyping. In *Proceedings of the 12th International Workshop on Implementation of Functional Languages*, number AIB-00-7 in Aachener Informatik Berichte, pages 279–294. RWTH Aachen, 2000.

[Hei95] N. Heintze. Control–flow analysis and type systems. In *Proceedings SAS '95, Second International Static Analysis Symposium, Glasgow, Scotland*, pages 189–206. Springer Lecture Notes in Computer Science, vol. 983, September 1995.

[Hen93] Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, 1993.

[HM97] Nevin Heintze and David McAllester. Linear-time subtransitive control flow analysis. In *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation*, number 32:6 in SIGPLAN notices, pages 261–272, June 1997.

[HRS95] Susan Horwitz, Thomas Reps, and Mooly Sagiv. Demand interprocedural dataflow analysis. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering, ACM SIGSOFT Software Engineering Notes 20, 4*, pages 104–115, 1995.

[Kae92] S. Kaes. Type inference in the presence of overloading, subtyping and recursive types. In *Proc. Conf. on LISP and Functional Programming*, 1992.

[Mit96] J.C. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.

[Mos96] Christian Mossin. *Flow Analysis of Typed Higher-Order Programs*. PhD thesis, DIKU, Department of Computer Science, University of Copenhagen, 1996.

[Mos98] Christian Mossin. Higher-order value flow graphs. *Nordic Journal of Computing*, 5:214–234, 1998.

[MR97] David Melski and Thomas Reps. Interconvertibility of set constraints and context-free language reachability. In *Proceedings PEPM*, pages 74–89, 1997.

[MR00] David Melski and Thomas Reps. Interconvertibility of set constraints and context-free language reachability. *Theoretical Computer Science*, 248, 2000. To appear.

[Myc84] A. Mycroft. Polymorphic type schemes and recursive definitions. In *Proceedings of the 6th International Symposium on Programming*, pages 217–228, 1984.

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

[OJ97] Robert O'Callahan and Daniel Jackson. Lackwit: A program understanding tool based on type inference. In *International Conference on Software Engineering*, May 1997.

[PO95] Jens Palsberg and Patrick O'Keefe. A type system equivalent to flow analysis. *Transactions on Programming Languages and Systems*, 17(4):576–599, July 1995.

[Pot96] F. Pottier. Simplifying subtyping constraints. In *International Conference on Functional Programming*, 1996.

[Reh97] J. Rehof. Minimal typings in atomic subtyping. In *Symposium on Principles of Programming Languages*, 1997.

[Reh98] J. Rehof. *The Complexity of Simple Subtyping Systems.* PhD thesis, Dept. of Computer Science, University of Copenhagen, Denmark, April 1998.

[Rep98] Thomas Reps. Program analysis via graph reachability. *Information and Software Technology*, 40 (11–12):701–726, 1998.

[Rep00] Thomas Reps. Undecidability of context-sensitive data-dependence analysis. *Transactions on Programming Languages and Systems*, 2000. to appear.

[RHS95] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Symposium on Principles of Programming Languages (POPL), San Francisco, California*, pages 49–61, 1995.

[Smi94] G. S. Smith. Principal type schemes for functional programs with overloading and subtyping. *Science of Computer Programming*, 23:197–226, 1994.

[TS96] V. Trifonov and S. Smith. Subtyping constrained types. In *Proceedings Static Analysis Symposium, Aachen, Germany*, 1996. LNCS 1145.

[Yan90] Mihalis Yannakakis. Graph-theoretic methods in database theory. In *Proceedings of the 9th ACM Symposium on Principles of Database Systems*, pages 230–242, 1990.

A Higher-order example

Figure 13 shows a higher-order example. The `app` function takes a function `f` as an argument and returns a function (labeled ℓ_r) that in turn takes a parameter `x` and applies `f` to `x`. The figure shows the flow graph resulting from applying `app` at instance i to the identity function `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 `app`, by observing what labels flow to ℓ_f. There is a path

$$\ell_{id} \xrightarrow{\mathsf{p}} \ell_8 \rightarrow \ell_9 \xrightarrow{\mathsf{n}} \ell_f$$

showing that the identity function (labeled ℓ_{id}) flows to ℓ_f. The flow edges connecting the types labeled by ℓ_8 and ℓ_9 arise from the subtype relation between the instance of `id` (the argument) and the domain of the instance of `app`. The reversed edge $\ell_1 \leq \ell_2$ arises through contra-variance of the subtype relation for function domains.

Edge $\ell_x \leq \ell_0$ represents the argument passing of `x` to `f` within `app`, and similarly, $\ell_6 \leq \ell_7$ represents the flow of the result of this application to the result of the function labeled ℓ_r. The flow path connecting `b` with `w` is then as follows:

$$\ell_b \rightarrow \bullet \xrightarrow{(_i} \ell_x \rightarrow \ell_0 \xrightarrow{)_i} \ell_1 \rightarrow \ell_2 \xrightarrow{(_j} \ell_y \rightarrow \ell_3 \xrightarrow{)_j} \ell_4 \rightarrow$$

$$\ell_5 \xrightarrow{(_i} \ell_6 \rightarrow \ell_7 \xrightarrow{)_j} \bullet \rightarrow \ell_w$$

Observe how the path enters `app` through instance i and then emerges back along the edge

$$\ell_0 \xrightarrow{)_i} \ell_1$$

The polarity of this edge was determined to be positive, because the polarity of the argument type ℓ_f within the type of `app` is itself negative. The path then traverses `id` on instance j and reenters `app` at instance i through

$$\ell_5 \xrightarrow{(_i} \ell_6$$

before finally emerging along the edge

$$\ell_7 \xrightarrow{)_i} \bullet$$

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 `app` twice through instance i.

B Complexity proof

Constraint generation 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 generate only $O(n)$ instantiation constraints. Our technical report [FRD00a] shows a variant of the algorithm presented here, producing only $O(n)$ constraints.

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 and avoid recomputing the vector $\vec{\ell'} = fl(A)$. The first problem is solved with an extra subsumption step on $\sigma_1 \leq \sigma_1'$ guaranteeing that all labels in σ_1' are fresh. Binding this type in place of σ_1 allows instantiation of all labels occurring in σ_1' at all instances. As for the second problem, labels $\vec{\ell'}$ can be accumulated incrementally in the abstraction rule [Lam] once and for all. The details of this are standard and can be found in [FRD00a].

Grammar 10 has m terminals $(_i,)_i$ 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.

Theorem 5.1 *Given I, C, ℓ, ℓ', answering a flow query $I; C \vdash_{\mathsf{CFL}} \ell \rightsquigarrow \ell'$ is decidable in time $O(n^3)$. Moreover, the entire flow relation derivable from I and C is computable in time $O(n^3)$.*

PROOF: The proof is in two steps. We first give a suboptimal bound to show that the generic algorithm in Figure 11 computes all derivable paths in time $O(en + un^2 + bn^3)$. Here e is the number of epsilon productions, u is the number of distinct unary grammar productions of the form $A \leftarrow B$, and b is the number of distinct binary grammar productions of the form $A \leftarrow B\ C$. Applied to our particular CFL problem (grammar of Figure 10) where $e = 3$, $u = 1$, and $b = 2m$, we obtain an initial complexity of $O(mn^3)$ for computing all pairs reachability. In the second step we tighten the complexity to $O(n^3)$ by exploiting the particular structure of constraints generated by $\mathsf{POLYFLOW}_{\mathsf{CFL}}$.

```
let app = (λf.(λx. f x)^{ℓr})^{ℓapp}
in
let id  = (λy.y)^{ℓid}
in
let w   = ((app^i id^j)^{ℓ'r} b^{ℓb})^{ℓw}
```

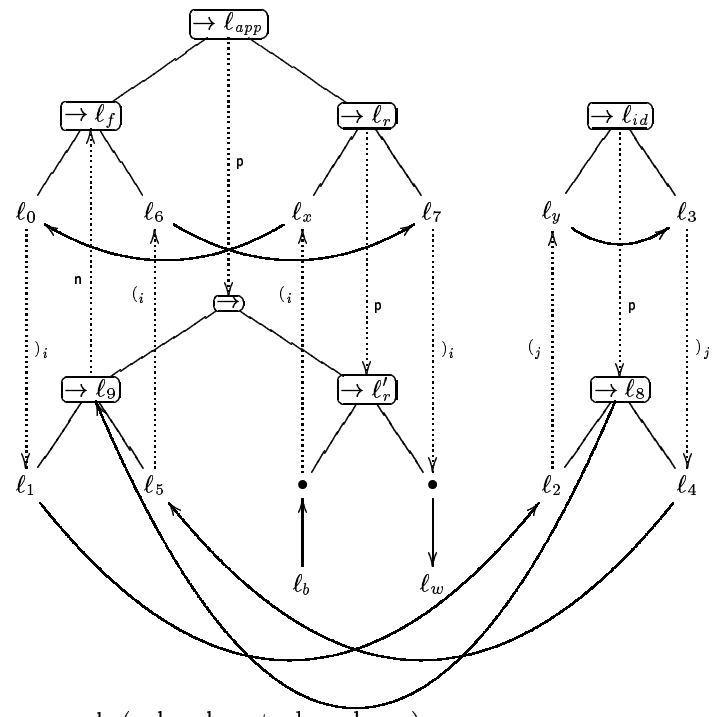


Figure 13: Higher-order example (only relevant edges shown)

We assume 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.

The algorithm uses a work list W and adds edges to the result graph G_0. For each node ℓ and each production r of the form $A \leftarrow B\ C$ of the grammar, we use two sets $pred_r(\ell)$ and $succ_r(\ell)$ containing the predecessors of ℓ reachable via an edge labeled B, and the successors of ℓ reachable via an edge labeled C.

Step 1. To see that the bound $O(en + un^2 + bn^3)$ holds, consider that 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 ℓ_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 ℓ_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 $A \leftarrow B\ C$. For a particular node ℓ_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 nodes ℓ_2, we obtain the bound $O(bn^3)$. The argument for productions of the form $A \leftarrow C\ B$ is analogous. The overall complexity bound of the algorithm is thus $O(en+un^2+bn^3)$.

Step 2. The complexity bound given above 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 $2m$ distinct productions).

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

If we discount the instantiation self-loops of the form $\ell \xrightarrow{\preceq_p^i} \ell$ added through rule [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 [Inst] is $O(m)$ per label in the worst case. Fortunately, the complexity analysis for general binary rules given above can be tightened due to the self-loop. Consider rule $M \leftarrow (_i\ K_i$, when applied to a self-loop. The situation is as follows:

$$(_i \ \ell \xleftarrow{K_i} \ell'$$

For a fixed ℓ and fixed i, the number of times this rule triggers is at most n, since there are at most n labels ℓ' connected to ℓ via an edge K_i. Since there are at most m such self-loops on ℓ and n distinct labels ℓ, 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 $K_i \leftarrow M\ )_i$, thereby proving the cubic bound of the theorem. □