Type-based flow analysis and context-free language reachability

MANUEL FÄHNDRICH and JAKOB REHOF

Mathematical Structures in Computer Science / Volume 18 / Special Issue 05 / October 2008, pp 823 - 894
DOI: 10.1017/S0960129508006968, Published online: 06 October 2008

Link to this article: http://journals.cambridge.org/abstract_S0960129508006968

How to cite this article:
MANUEL FÄHNDRICH and JAKOB REHOF (2008). Type-based flow analysis and context-free language reachability. Mathematical Structures in Computer Science, 18, pp 823-894 doi:10.1017/S0960129508006968

Math. Struct. in Comp. Science (2008), *vol.* 18, *pp.* 823–894. © 2008 Cambridge University Press
doi:10.1017/S0960129508006968 Printed in the United Kingdom

Type-based flow analysis and context-free language reachability

MANUEL FÄHNDRICH[†] and JAKOB REHOF

One Microsoft Way, Redmond WA 98052, U.S.A.
[†]*Email:* maf@microsoft.com

Received 15 February 2006; revised 17 January 2007

We present a novel approach to computing the 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 $O(n^8)$ 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

1.1. *Type-based flow analysis*

Type-based program analyses have received much attention (see, for example, Palsberg and O'Keefe (1995), Heintze (1995), Mossin (1996), Heintze and McAllester (1997), Nielson *et al.* (1999) and Fähndrich *et al.* (2000)). Attractive properties of such analyses include: they typically work directly on higher-order programs with structured data; they provide a natural separation between the specification (type system) and implementation of the analysis; and standard techniques from type theory are applicable for reasoning about properties (for example, soundness and 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, for example, Mossin (1996) and Nielson *et al.* (1999)). 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 for answering 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*, that is, it avoids spurious flow between different calling contexts. Subtyping further enhances analysis precision by modelling a *directional* (non-symmetric) notion of value flow, see, for example, Heintze (1995).

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 (Henglein 1993)). Based on this presentation, we are able to apply *Context-Free Language (CFL) Reachability* (Reps *et al.* 1995) 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 (Mossin 1996) based on polymorphic subtyping from $O(n^8)$ to $O(n^3)$, where n is the size of the typed program (n can theoretically be exponential in program size but is close to program size in practice (Mitchell 1996)).

— 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. Also, our algorithm leads to demand-driven techniques, which, to our knowledge, have not been obtained before for polymorphic subtyping.

In the paper Fähndrich *et al.* (2000) we presented a flow analysis in polymorphic type systems based on instantiation constraints but without subtyping. The work reported in Rehof and Fähndrich (2001) provides a substantial generalisation of Fähndrich *et al.* (2000) through the incorporation of subtyping. The present paper is an extended version of Rehof and Fähndrich (2001). The extensions include: a full proof of soundness for the flow analysis presented in Rehof and Fähndrich (2001); several refined versions of the algorithm for computing flow information; more examples; and much greater in-depth discussion of related systems. In particular, we study polymorphism both at the level of flow labels and at the level of the underlying type structure, and we show how the technique presented in Fähndrich *et al.* (2000) arises as a systematic specialisation of the general framework first presented in Rehof and Fähndrich (2001).

1.2. *Overview*

This paper is organised as follows. In Section 2 we start with the system POLYFLOW, which is a flow type system with polymorphic recursion, allowing for 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 (Mossin 1996) based on constraint copying and simplification ($\mathsf{POLYFLOW}_{\mathsf{copy}}$) is equivalent to a system with neither copying nor simplification based on instantiation constraints and context-free language reachability queries ($\mathsf{POLYFLOW}_{\mathsf{CFL}}$). The advantages of $\mathsf{POLYFLOW}_{\mathsf{CFL}}$ over $\mathsf{POLYFLOW}_{\mathsf{copy}}$ 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 in Reps *et al.* (1995). Furthermore, we provide insights into the complexity analysis of CFL-based algorithms through a new CFL algorithm.

Section 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 (Mycroft 1984) type system. We skip the copying system and study directly the system $\mathsf{POLYTYPE}_{\mathsf{CFL}}$ based on instantiation constraints and CFL-reachability. The novel aspect of $\mathsf{POLYTYPE}_{\mathsf{CFL}}$ 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 synchronised at instantiation boundaries. This synchronisation is possible because we work with a structural subtyping problem without recursive types. Perhaps surprisingly, the complexity of computing flow for $\mathsf{POLYTYPE}_{\mathsf{CFL}}$ is no harder than in $\mathsf{POLYFLOW}_{\mathsf{CFL}}$: individual or all queries are computable in time $O(n^3)$. However, the two complexities differ in the way the number of nodes n varies with the size of the program m. In the worst case for $\mathsf{POLYFLOW}_{\mathsf{CFL}}$ n is exponential in m, whereas for $\mathsf{POLYTYPE}_{\mathsf{CFL}}$, 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 $\mathsf{POLYTYPE}_{\mathsf{CFL}}$ 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 $\mathsf{POLYFLOW}_{\mathsf{CFL}}$ with a more efficient implementation. Individual

$\ell \in$ Labels

$\tau \in$ Types

$$\tau \ ::= \ int \mid \tau \rightarrow \tau \mid \tau \times \tau$$

$\sigma \in$ Labelled Types

$$\sigma \ ::= \ int^{\ell} \mid \sigma \rightarrow^{\ell} \sigma \mid \sigma \times^{\ell} \sigma$$

$e \in$ Terms

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

Fig. 1. Definitions

flow queries can be answered in linear time, and all queries in quadratic time. We explain how the technique described in Fähndrich *et al.* (2000) arises from the general CFL-based approach of Rehof and Fähndrich (2001) in a systematic way, by specialisation.

Sections 2 and 3 are concerned with the logical aspects of the flow systems under study and only show 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 using algorithm *W* for Hindley–Milner typings or the semi-algorithm of Henglein (Henglein 1993) for Milner–Mycroft typings. In Section 4, we give both a discussion of related work and related systems, and consider open problems. Section 5 presents conclusions, and Appendix A provides a proof of soundness for the flow relation of $\mathsf{POLYFLOW}_{\mathsf{CFL}}$.

1.3. *Definitions*

Throughout this article, we study the simple ML-like language defined in Figure 1. 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 and letrec bound variables as well as on instances of let and letrec bound variables. Each sub-expression in the source is implicitly labelled, as in e^{ℓ}. However, we omit this labelling unless 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 unlabelled types τ and labelled types σ. As a technical device, we use the notation $\tau\langle s\rangle$ to denote labelled types with structure τ. The *label sequence* s is the labelling of τ obtained as a pre-order traversal of its structure. Figure 2 makes the relation between $\tau\langle s\rangle$ and σ precise. A judgment $\vdash \tau\langle s\rangle : \sigma$ states that type $\tau\langle s\rangle$ is *well labelled* and equal to σ. When σ is not of interest, we will also write $\vdash \tau\langle s\rangle\ \mathsf{wl}$ for such statements. We use well-labelled types $\tau\langle s\rangle$ interchangeably with σ.

$\boxed{\vdash \tau\langle s\rangle : \sigma}$

$$\frac{}{\vdash int\langle \ell\rangle : int^{\ell}}$$

$$\frac{\vdash \tau_1\langle s_1\rangle : \sigma_1 \qquad \vdash \tau_2\langle s_2\rangle : \sigma_2}{\vdash \tau_1 \times \tau_2\langle \ell s_1 s_2\rangle : \sigma_1 \times^{\ell} \sigma_2} \qquad \frac{\vdash \tau_1\langle s_1\rangle : \sigma_1 \qquad \vdash \tau_2\langle s_2\rangle : \sigma_2}{\vdash \tau_1 \to \tau_2\langle \ell s_1 s_2\rangle : \sigma_1 \to^{\ell} \sigma_2}$$

Fig. 2. Well labelling

Positive and negative polarity

We say that τ_0 has positive polarity in τ if one of the following applies:

1 $\tau = \tau_0$;
2 $\tau = \tau_1 \to \tau_2$ and either τ_0 has positive polarity in τ_2 or τ_0 has negative polarity in τ_1;
3 $\tau = \tau_1 \times \tau_2$ and τ_0 has positive polarity in τ_1 or τ_2.

Similarly, τ_0 has negative polarity in τ, if either:

1 $\tau = \tau_1 \to \tau_2$ and either τ_0 has negative polarity in τ_2 or τ_0 has positive polarity in τ_1; or
2 $\tau = \tau_1 \times \tau_2$ and τ_0 has negative polarity in τ_1 or τ_2.

The definition is identical for labelled types σ. A label annotating a type τ_0 with positive (negative) polarity within σ itself has positive (negative) polarity. We also call a label with negative polarity within σ an *input label*, and one with positive polarity an *output label*. The set of labels with positive polarity in σ are written σ^{+} and the set of labels with negative polarity in σ is written $\sigma^{\div}$.

Contexts

We assign each subexpression e' of an expression e a context Q as follows. The context of e_1 in an expression $\texttt{let}^{\ell}\ f\ \texttt{=}\ e_1\ \ldots$ is the label ℓ of the let expression. The context of e_1 in $\texttt{letrec}^{\ell}\ f\ \texttt{=}\ e_1\ \ldots$ is the label ℓ 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 $\ell \leqslant \ell$. Statements of the form $C \vdash \ell \leqslant \ell'$ express the fact that C implies $\ell \leqslant \ell'$. These statements are governed by the rules in Figure 3. The generalised form $C \vdash C'$ abbreviates $C \vdash \ell \leqslant \ell'$ for each $\ell \leqslant \ell'$ in C'.

Structural subtyping

Subtyping judgments relate two labelled types and have the form $C \vdash \sigma \leqslant \sigma'$. In this paper we only consider structural subtyping, that is, σ and σ' have the same structure

$\boxed{C \vdash \ell \leqslant \ell}$

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

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

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

Fig. 3. Constraint relation

$\boxed{C \vdash^{\leqslant} \sigma \leqslant \sigma}$

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

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

$$\frac{C \vdash^{\leqslant} \sigma_1' \leqslant \sigma_1 \qquad C \vdash^{\leqslant} \sigma_2 \leqslant \sigma_2' \qquad C \vdash \ell \leqslant \ell'}{C \vdash^{\leqslant} \sigma_1 \to^{\ell} \sigma_2 \leqslant \sigma_1' \to^{\ell'} \sigma_2'}\,[\text{Fun}]$$

Fig. 4. Subtype relation

τ. The subtyping is restricted to the labelling as given by the rules in Figure 4. As is standard, the function type constructor is contravariant in the domain.

Substitutions

Substitutions φ map labels to labels. The domain of a substitution φ is the set of labels on which φ 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 in the natural way.

Instantiation constraints

An instantiation constraint $\ell \preceq_p^i \ell'$ 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. Unlike the case for 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 5 lifts instantiations to labelled types.

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

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

$$\frac{}{I, \ell \preceq_p^i \ell' \vdash \ell \preceq_p^i \ell'}\,[\mathrm{Id}] \qquad \frac{I \vdash \ell \preceq_p^i \ell'}{I \vdash int^{\ell} \preceq_p^i int^{\ell'}}\,[\mathrm{Int}]$$

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

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

Fig. 5. Instantiation relation

constraint $\ell \preceq_p^i \ell'$ in I. A set I of instantiation constraints thus gives rise to a vector of substitutions $\Phi(I)$ indexed by instantiation site i, where

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

We use the notation $I \vdash \sigma \preceq_p^i \sigma' : \varphi$ to mean that judgment $I \vdash \sigma \preceq_p^i \sigma'$ is derivable using the rules of Figure 5, and that $\varphi(\sigma) = \sigma'$. Note that φ only has to agree with $\Phi_i(I)$ on the labels appearing in σ.

1.4. *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 1.4.1 and Section 1.4.2). We then give, in Sections 1.4.3 and 1.4.4, an intuitive overview of our solution.

We illustrate our techniques with a very simple example program e:

```
let id = $\lambda$x:$int^{\ell_1}$.x$^{\ell_2}$
in
   ((id$^i$ 0$^{\ell_3}$)$^{\ell_4}$, (id$^j$ 1$^{\ell_5}$)$^{\ell_6}$)
end
```

Here we are interested in tracking the flow of constants 0 and 1 labelled with ℓ_3 and ℓ_5.

1.4.1. *Problem 1: Constraint copying*

Prior to Rehof and Fähndrich (2001), all work in polymorphic subtype inference was based on *qualified polymorphic types* of the form $\forall\vec{\ell}.C \Rightarrow \sigma$, where C is a set of captured subtyping constraints 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. A standard[†] polymorphic constrained type (see, for example, Smith (1994), Trifonov and Smith (1996) and Mossin (1996)) for

[†] Function id can be given a most general typing without any subtyping constraints, but we choose the present typing for illustrative purposes.

`id` is $\forall \ell_1 \ell_2 . \{\ell_1 \leqslant \ell_2\} \Rightarrow int^{\ell_1} \rightarrow int^{\ell_2}$. In a *copy-based framework*, program e is typed by copying the constraint set $\{\ell_1 \leqslant \ell_2\}$ associated with `id` at each of the instantiation sites $\mathtt{id}^i$ and $\mathtt{id}^j$, yielding the standard polymorphic typing judgment

$$\{\ell_3 \leqslant \ell_4, \ell_5 \leqslant \ell_6\}; \varnothing \vdash e : int^{\ell_4} \times int^{\ell_6} . \tag{1}$$

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 conclude that the value `0` (ℓ_3) flows to the first component of the resulting pair (ℓ_4), and the value `1` (ℓ_5) flows to the second (ℓ_6). Polymorphism over labels, here implemented through constraint copying, keeps the two instantiation sites apart, matching up a call site (for example, $\mathtt{id}^i\ \mathtt{0}^{\ell_3}$) with its proper return (ℓ_4). 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 apparent 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†. The problem has generated a significant amount of research, including work on *constraint simplification*, which aims to compact constraint sets before they are copied (Fuh and Mishra 1989; Curtis 1990; Kaes 1992; Smith 1994; Eifrig *et al.* 1995; Pottier 1996; Trifonov and Smith 1996; Fähndrich and Aiken 1996b; Aiken *et al.* 1997; Rehof 1997; Flanagan and Felleisen 1997). It is unlikely that constraint simplification techniques alone will solve this problem, and complete simplification is a hard problem itself (Rehof 1998; Flanagan and Felleisen 1997).

1.4.2. *Problem 2: Demand-driven flow computation* The constraint copying methods for polymorphic subtyping systems are not demand driven (Mossin 1996). 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 $\{\ell_1 \leqslant \ell_2\}$ 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, and 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). 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 non-trivial (Mossin 1996; Foster *et al.* 2000), both in terms of computational expense and correctness. Furthermore, it is unclear how such techniques can accommodate demand-driven versions naturally.

1.4.3. *A new method based on instantiation constraints* Both of the problems mentioned in Section 1.4.1 and 1.4.2 inhibit scalability of polymorphic subtype-based flow analysis. We tackle these problems by introducing a new presentation of polymorphic subtyping, based on *instantiation constraints* (Fähndrich *et al.* 2000). We call this system $\mathsf{POLYFLOW}_{\mathsf{CFL}}$.

† Of course, the small size of the constraint set in our toy example is illusory in practice.

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. Term e from the example receives a typing of the form

$$\left\{ \begin{array}{l} \ell_1 \preceq^i_{\div} \ell_3, \ell_2 \preceq^i_{+} \ell_4, \\ \ell_1 \preceq^j_{\div} \ell_5, \ell_2 \preceq^j_{+} \ell_6 \end{array} \right\} ; \{\ell_1 \leqslant \ell_2\} ; \varnothing \vdash e : \sigma \tag{2}$$

where $\sigma = int^{\ell_4} \times int^{\ell_6}$. A judgment in $\mathsf{POLYFLOW}_{\mathsf{CFL}}$ has six components: these are, 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 $\varphi_i = \{\ell_1 \mapsto \ell_3, \ell_2 \mapsto \ell_4\}$ and $\varphi_j = \{\ell_1 \mapsto \ell_5, \ell_2 \mapsto \ell_6\}$ that are used at instantiation sites i and j, respectively, to produce instances of `id`'s type $int^{\ell_1} \to int^{\ell_2}$. Here, 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 $\{\ell_1 \leqslant \ell_2\}$ associated with `id`, yielding $\{\ell_3 \leqslant \ell_4\}$ and $\{\ell_5 \leqslant \ell_6\}$. The *indices* i and j serve to keep substitutions at distinct instantiation sites apart.

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

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

Having addressed the copy problem from Section 1.4.1, 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 we demand to know where ℓ_3 flows. Drawing flow constraint $\{\ell_1 \leqslant \ell_2\}$ 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:

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

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 instantiation edge $\ell_1 \preceq^i_{\div} \ell_3$ represents the flow of an actual to a formal, and 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 directions. Negative $\div$ instantiation edges represent flow in the direction opposite to 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 and ℓ_2 positively in the type $int^{\ell_1} \rightarrow int^{\ell_2}$ of `id`. Instantiation polarities were introduced in Fähndrich *et al.* (2000). 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 the parts of the constraint systems needed for this query. This addresses the on-demand problem from Section 1.4.2.

A further advantage of instantiation constraints, which addresses the second problem in Section 1.4.2, is that *all* flow is present in the constraint sets I and C obtained in a typing judgment for $\mathsf{POLYFLOW_{CFL}}$. 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 in a similar way to the flow above using $\ell_1 \preceq^{i}_{\div} \ell_3$ and $\ell_1 \preceq^{j}_{\div} \ell_5$. To recover such flow in copy-based systems, the entire typing derivation is required instead of merely the final judgment.

1.4.4. *CFL-reachability* How can we systematically recover the global flow from instantiation and flow constraints? As we will show in this article, this is best formulated as *context-free language (CFL) reachability* on a graph formed by flow and instantiation constraints (edges) and labels (nodes). We will now present the intuition behind this idea. We assume that in this graph: negative instantiation constraints are reversed and labelled with opening parentheses $(_i$; positive instantiation edges are labelled closing parentheses $)_i$; and flow edges are labelled with d. All flow paths in the graph now spell words. For example, the path $\ell_3 \xrightarrow{(_i} \ell_1 \xrightarrow{\mathsf{d}} \ell_2 \xrightarrow{)_i} \ell_4$ 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 flows 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 characterised 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 Reps *et al.* (1995) for the case of first-order programs manipulating atomic data. We incorporate higher-order functions and structured data. Here it is important to notice that a function symbol in higher-order programs may occur in contexts that are not call sites, and matched flow may not correspond to actual calls and returns.

2. Flow polymorphism

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

2.1. *A copy-based system*

POLYFLOW$_{\mathsf{copy}}$ is similar to the system Mossin studied in his thesis (Mossin 1996, Chapter 5). However, it differs in the following respects:

— Mossin's approach can only answer 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$_{\mathsf{copy}}$ is slightly more general in that it allows all queries of the form 'Is there flow from program point ℓ_1 to ℓ_2?'. As a result, we do not require a class of constant labels.
— We make instantiation substitutions explicit through instantiation constraints.

Our use of instantiation constraints, while not strictly necessary for the presentation of a copy-based system, facilitate the transition to the system POLYFLOW$_{\mathsf{CFL}}$ by making the substitutions involved in copying explicit. Hence, POLYFLOW$_{\mathsf{copy}}$ is a presentational variant of Mossin's system and enjoys the same basic properties, including evaluation order independence and modularity. For further comparison with Mossin's system see Mossin (1996, Chapter 5).

2.1.1. *Polymorphic constrained types* Polymorphic types for POLYFLOW$_{\mathsf{copy}}$ are qualified by a set of constraints, written $\forall\vec{\ell}.C \Rightarrow \sigma$. Such a type scheme can be instantiated to $\sigma[\vec{\ell'}/\vec{\ell}]$ in all contexts where the constraints $C[\vec{\ell'}/\vec{\ell}]$ 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 σ. These constraints summarise *all* flow between input and output labels of σ.

We use the notation $fl(\sigma)$ for the set of labels in σ and $fl(A)$ for the set of free labels in the environment A. We may also use this notation on flow constraints $fl(C)$.

2.1.2. *Type rules* Figure 6 contains typing rules for deriving judgments of the form $C_g; I; C; A \vdash_{cp} e : \sigma$, where:

— C_g is an accumulation of all flow constraints appearing in the derivation but not in C;
— I is a set of instantiation constraints;
— C is a set of flow constraints;
— A is a type environment mapping lambda bound variables x to labelled types σ, 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 τ to obtain the labelled type $\tau\langle s\rangle$ used in the judgment of the lambda body. Rule [Label] connects the label ℓ annotating an expression e^{ℓ} with the label ℓ' annotating the top-level constructor of the type $\tau\langle\ell' s\rangle$ of e.

$$\boxed{C_g;I;C;A \vdash_{cp} e : \sigma}$$

Base rules

$$\frac{}{C_g;I;C;A,x:\sigma \vdash_{cp} x : \sigma}\,[\mathrm{Id}] \qquad \frac{}{C_g;I;C;A \vdash_{cp} n^{\ell} : int^{\ell}}\,[\mathrm{Int}]$$

$$\frac{C_g;I;C;A \vdash_{cp} e_1 : \sigma_2 \to^{\ell} \sigma_1 \qquad C_g;I;C;A \vdash_{cp} e_2 : \sigma_2}{C_g;I;C;A \vdash_{cp} e_1\ e_2 : \sigma_1}\,[\mathrm{App}]$$

$$\frac{\vdash \tau\langle s\rangle : \sigma \qquad C_g;I;C;A,x:\sigma \vdash_{cp} e : \sigma'}{C_g;I;C;A \vdash_{cp} \lambda^{\ell}x{:}\tau.e : \sigma \to^{\ell} \sigma'}\,[\mathrm{Lam}]$$

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

$$\frac{C_g;I;C;A \vdash_{cp} e : \sigma_1 \times^{\ell} \sigma_2}{C_g;I;C;A \vdash_{cp} e.j : \sigma_i}\,[\mathrm{Proj}\ j = 1,2]$$

$$\frac{C_g;I;C;A \vdash_{cp} e_0 : int^{\ell} \qquad C_g;I;C;A \vdash_{cp} e_1 : \sigma \qquad C_g;I;C;A \vdash_{cp} e_2 : \sigma}{C_g;I;C;A \vdash_{cp} \texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \sigma}\,[\mathrm{Cond}]$$

$$\frac{\begin{array}{l} C_g;I;C;A \vdash_{cp} e : \sigma \\ C \vdash \sigma \leqslant \sigma' \end{array}}{C_g;I;C;A \vdash_{cp} e : \sigma'}\,[\mathrm{Sub}] \qquad \frac{\begin{array}{l} C_g;I;C;A \vdash_{cp} e : \sigma \\ \vdash \tau\langle \ell' s\rangle : \sigma \qquad C \vdash \ell = \ell' \end{array}}{C_g;I;C;A \vdash_{cp} e^{\ell} : \sigma}\,[\mathrm{Label}]$$

Polymorphic rules

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

$$\frac{\begin{array}{l} C_g;I;C';A,f:\forall\vec{\ell}.C' \Rightarrow \sigma_1 \vdash_{cp} e_1 : \sigma_1 \\ C_g;I;C;A,f:\forall\vec{\ell}.C' \Rightarrow \sigma_1 \vdash_{cp} e_2 : \sigma_2 \\ C' \subseteq C_g \qquad \vec{\ell} = gen(A,\sigma_1,C') \qquad \vdash \tau\langle s\rangle : \sigma_1 \end{array}}{C_g;I;C;A \vdash_{cp} \texttt{letrec}\ f{:}\tau\ \texttt{=}\ e_1\ \texttt{in}\ e_2 : \sigma_2}\,[\mathrm{Rec}]$$

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

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

Fig. 6. Copy-based system $\mathsf{POLYFLOW}_{\mathsf{copy}}$

The statement $C \vdash \ell = \ell'$ is an abbreviation for the statements $C \vdash \ell \leqslant \ell'$ and $C \vdash \ell' \leqslant \ell$. Tying the expression label ℓ to the type label ℓ' in such a way allows us to ask our flow queries on the expression labels, but to 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 $\forall\vec{\ell}.C' \Rightarrow \sigma_1$. The set of quantified labels $\vec{\ell}$ contains all free labels of σ_1 and C' that are not free in A. Furthermore, C' must be a subset of the collection C_g. Rule [Inst] gives types to occurrences f^i of let and letrec bound variables. The polymorphic type $\forall\vec{\ell}.C' \Rightarrow \sigma$ is instantiated to σ' through a substitution φ on the quantified labels $\vec{\ell}$. The judgment $I \vdash \sigma \preceq^i_+ \sigma' : \varphi$ requires that the substitution of labels in σ is explicit as a set of instantiation constraints in I. Recall that this judgment only relates I and φ for labels actually occurring in σ. 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 σ_1 for e_1 under constraints C' and the assumption that f's type is $\forall\vec{\ell}.C' \Rightarrow \sigma_1$. As in the [Let] rule, we require that C' is a subset of the collection C_g.

Contexts of labels We assume that in the derivation of a judgment $C_g; I; C; A \vdash_{cp} e : \sigma$, all labels of sub-expressions of e are distinct, and that quantified labels of distinct quantified types are distinct through the renaming of bound labels. Recall from Section 1.3 that let and letrec bindings divide the sub-expressions of e into a set of disjoint contexts. We associate each label ℓ appearing in the derivation with a unique context as follows. The context of ℓ 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 by means of: a label (its context); a let or letrec binding; or a particular constraint 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 ℓ.

To gain some intuition about contexts, if we suppose that all let and letrec bindings generalise lambda expressions, and all lambda's are let or letrec bound, then contexts coincide with lambda bodies.

2.1.3. *Flow graphs*

A *flow graph* $G = (I, C_g \cup C, L)$ of a derivation $C_g; I : C; A \vdash_{cp} e : \sigma$ 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 $C_g \cup C$.

Figure 7 shows an example flow graph. Solid edges represent flow constraints and dotted edges represent instantiation constraints. Large circles group nodes of the same context. Instantiation edges usually cross between 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.

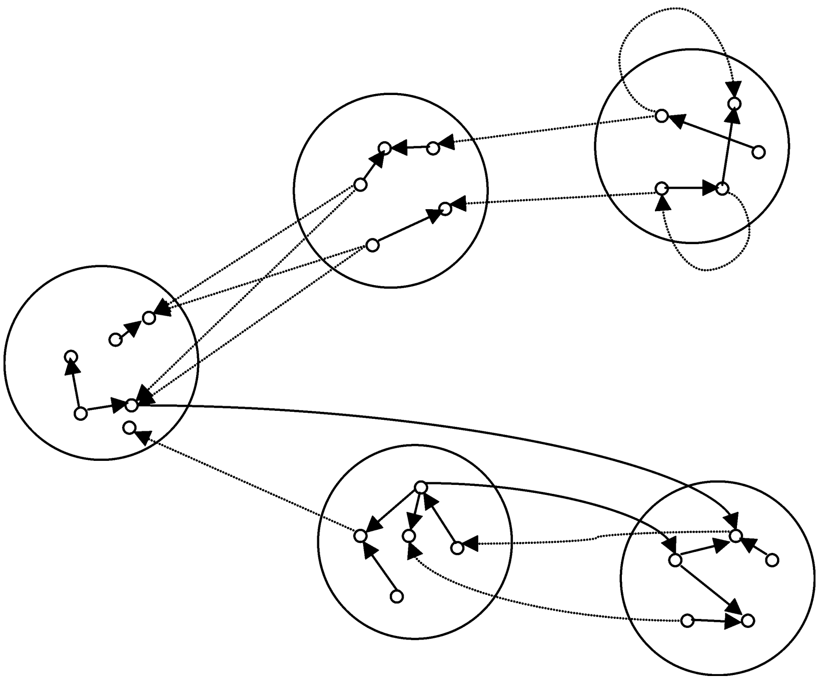


Fig. 7. Generic flowgraph with five contexts

Answering a flow query of the form ‘Is there flow from ℓ_1 to ℓ_2’ requires finding a path in a flow graph from ℓ_1 to ℓ_2 involving flow and instantiation edges. In the special case where ℓ_1 and ℓ_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 the answering of flow queries precise using a simple flow relation.

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 $C_g \cup C$. Figure 8 contains a set of rules for deriving *flow judgments* of the form $I;C_g \cup C \vdash_{cp} \ell_1 \rightsquigarrow \ell_2$ stating that under constraints I and $C_g \cup C$, there is flow from label ℓ_1 to label ℓ_2. We now examine these rules in turn.

Some flow paths only involve flow constraints in $C_g \cup C$. These can be deduced directly using rule [Level], that is, if we can derive $C_g \cup C \vdash \ell_1 \leqslant \ell_2$, we can deduce the flow from ℓ_1 to ℓ_2.

Note that internally the rules use two auxiliary judgments of the form $I;C \vdash \ell_1 \rightsquigarrow_+ \ell_2$ and $I;C \vdash \ell_1 \rightsquigarrow_\div \ell_2$, which are called *positive* and *negative flow*, respectively. Rules with an occurrence of p are actually rule schemes for all rules obtained by selecting $+$ or $\div$ for p.

$\boxed{I\,;C \vdash_{cp} \ell \rightsquigarrow \ell}$

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

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

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

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

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

$p = +, \div$

Fig. 8. Flow relation for $\mathsf{POLYFLOW}_{\mathsf{copy}}$

Rule [Trans] encodes transitivity for both auxiliary relations. The interesting rules are [In] [Out] and [Stage]. These rules capture the fact 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 9 contains an example program and flow graph. Function `idpair` is the identity on integer pairs. It is instantiated at site i within `f`, which is, in turn, instantiated at site j. There are three distinct contexts C_0, C_1 and C_2 corresponding to the body of `idpair`, the body of `f` and the remainder of the code. The dashed flow edges in C_1 arise through copying of the corresponding edges from C_0 at site i. Using the flow relation of Figure 8, we can deduce flow $I\,;C_g \vdash_{cp} \ell_b \rightsquigarrow \ell_z$ 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 ℓ_b, giving rise to $I\,;C_g \vdash \ell_b \rightsquigarrow_+ \ell_1$. Combined using [Trans] with flow $I\,;C_g \vdash \ell_1 \rightsquigarrow_+ \ell_z$ through [Out], this yields $I\,;C_g \vdash \ell_b \rightsquigarrow_+ \ell_z$. Finally, using [Stage] together with a trivial negative flow $I\,;C_g \vdash \ell_z \rightsquigarrow_\div \ell_z$ derived from [Level] results in $I\,;C_g \vdash_{cp} \ell_b \rightsquigarrow \ell_z$.

```
let idpair = λx:int × int.x in
let f      = λy:int.idpair^i (a^ℓa,b^ℓb) in
let z      = (f^j 0).2^ℓz
...
```

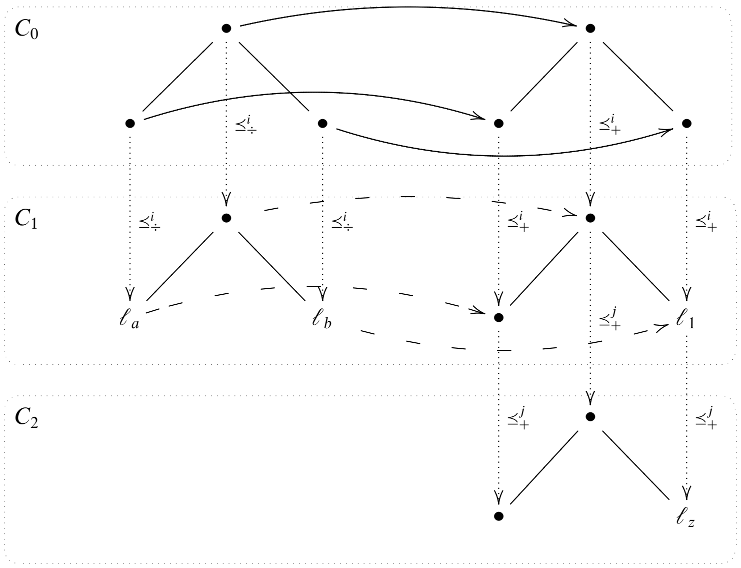


Fig. 9. POLYFLOW$_{\mathsf{copy}}$ example

2.2. *A CFL-based system*

We now present the system POLYFLOW$_{\mathsf{CFL}}$ that computes the same answers to flow queries as POLYFLOW$_{\mathsf{copy}}$, but does not involve any constraint copying. POLYFLOW$_{\mathsf{CFL}}$ is conceptually simpler than POLYFLOW$_{\mathsf{copy}}$, and also more practical. The best-known algorithm (Mossin 1996) for inferring a type derivation for POLYFLOW$_{\mathsf{copy}}$ has complexity $O(n^8)$ and involves handling multiple constraint systems, their simplification, and copying. On the other hand, POLYFLOW$_{\mathsf{CFL}}$ involves only a single flow constraint system. Computing a type derivation and answering all flow queries can be done in cubic time for POLYFLOW$_{\mathsf{CFL}}$. We will formally prove the soundness of flow queries in POLYFLOW$_{\mathsf{CFL}}$ with respect to POLYFLOW$_{\mathsf{copy}}$.

2.2.1. *Polymorphic types*

POLYFLOW$_{\mathsf{CFL}}$ uses polymorphic types of the form $\forall\vec{\ell}.\sigma$. Only labels are quantified, and unlike for POLYFLOW$_{\mathsf{copy}}$, no qualifying constraint systems appear in the polymorphic types.

2.2.2. *Type rules*

Judgments for POLYFLOW$_{\mathsf{CFL}}$ have the form $t;I;C;A \vdash_{\mathsf{CFL}} e : \sigma$ shown in Figure 10. The base rules change minimally with respect to the copy system. The extra t component in the judgment is a label sequence containing the labels appearing in all λ-bound types in the environment A. The [Lam] rule makes this explicit by concatenating

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

Base rules

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

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

$$\frac{\vdash \tau\langle s \rangle : \sigma \qquad t\,s; I; C; A, x : \sigma \vdash_{\mathsf{CFL}} e : \sigma'}{t; I, C; A \vdash_{\mathsf{CFL}} \lambda^{\ell} x{:}\tau.e : \sigma \to^{\ell} \sigma'}\,[\text{Lam}]$$

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

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

$$\frac{t; I; C; A \vdash_{\mathsf{CFL}} e_0 : int^{\ell} \qquad t; I; C; A \vdash_{\mathsf{CFL}} e_1 : \sigma \qquad t; I; C; A \vdash_{\mathsf{CFL}} e_2 : \sigma}{t; I; C; A \vdash_{\mathsf{CFL}} \texttt{if0}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \sigma}\,[\text{Cond}]$$

$$\frac{t; I; C; A \vdash_{\mathsf{CFL}} e : \sigma \qquad C \vdash \sigma \leqslant \sigma'}{t; I; C; A \vdash_{\mathsf{CFL}} e : \sigma'}\,[\text{Sub}] \qquad \frac{\begin{array}{c} t; I; C; A \vdash_{\mathsf{CFL}} e : \sigma \\ \vdash \tau\langle \ell' s \rangle : \sigma \qquad C \vdash \ell = \ell' \end{array}}{t; I; C; A \vdash_{\mathsf{CFL}} e^{\ell} : \sigma}\,[\text{Label}]$$

Polymorphic rules

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

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

$$\frac{I \vdash \sigma \preceq^{i}_{+} \sigma' : \varphi \qquad dom\,\varphi = \vec{\ell} \qquad I \vdash s \preceq^{i}_{+} s \qquad I \vdash s \preceq^{i}_{\div} s}{t; I; C; A, f : (\forall \vec{\ell}.\sigma, s) \vdash_{\mathsf{CFL}} f^{i} : \sigma'}\,[\text{Inst}]$$

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

Fig. 10. CFL-based system $\mathsf{POLYFLOW}_{\mathsf{CFL}}$

the label sequence s of the new λ-bound type $\tau\langle s \rangle$ to the current sequence t for the judgment of the body. The labels of λ bound variables have a special role in flow computations. Due to the nesting of let and letrec bound definitions within lambda expressions, λ bound program variables allow the program direct access to non-local scopes.

Intuitively[†], such edges skip some call or return edges that are manifested as instantiation edges. In this section we transform the flow problem into a CFL-reachability problem by formulating matched call–return flows as a parenthesis matching problem. Cross-context flow edges mean we need 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 let and letrec bindings, we capture the current set of free labels as part of the bindings. For letrec and let bound variables, the environment contains pairs of the polymorphic type and a label sequence: $A, f : (\forall \ell_i.\sigma, s)$. At instantiation points i [Inst], we require all bound variables in the type $\forall \vec{\ell}.\sigma$ to be uniquely named across the derivation (program), and we require instantiations $I \vdash s \preceq^i_+ s$ and $I \vdash s \preceq^i_\div s$ that have the effect that for all ℓ in s, there are constraints $\ell \preceq^i_+ \ell$ and $\ell \preceq^i_\div \ell$ in I. We will show later (Section 2.2.7) how these so-called *self-loops* enter into the flow computation.

2.2.3. *Flow graphs and flow queries* Given a derivation $\cdot; I; C \vdash_{\mathsf{CFL}} e : \sigma$, 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.

The flow relation of $\mathsf{POLYFLOW}_{\mathsf{copy}}$ is basically transitivity on the positive and negative flow relations $\rightsquigarrow_+$ and $\rightsquigarrow_\div$ involving constraints of C_g. Since we do not copy any constraints in the $\mathsf{POLYFLOW}_{\mathsf{CFL}}$ 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 $\mathsf{POLYFLOW}_{\mathsf{copy}}$. The basic case where $\mathsf{POLYFLOW}_{\mathsf{copy}}$ has an explicit flow $\ell'_1 \rightsquigarrow_p \ell'_2$ not present in $\mathsf{POLYFLOW}_{\mathsf{CFL}}$ appears in the following situation:

$$\begin{tikzcd}
\ell_1 \arrow[r, bend left] \arrow[d, "\preceq^i_\div"] & \ell_2 \arrow[d, "\preceq^i_+"] \\
\ell'_1 \arrow[r, dotted] & \ell'_2
\end{tikzcd}$$

$\mathsf{POLYFLOW}_{\mathsf{copy}}$ generates the edge $\ell'_1 \leqslant \ell'_2$ by copying the constraint $\ell_1 \leqslant \ell_2$ at instance i. In $\mathsf{POLYFLOW}_{\mathsf{CFL}}$, the flow from ℓ'_1 to ℓ'_2 must be discovered through other means. To do so, we introduce an auxiliary flow judgment $C \vdash \ell_1 \rightsquigarrow_{\mathsf{m}} \ell_2$ called *matched* flow. Matched flow captures flow paths where instantiation edges form matching pairs: negative instantiation 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 ℓ_1 to ℓ_2, rule [Match] in Figure 11 can be used to deduce matched flow from ℓ'_1 to ℓ'_2, effectively deducing the copied constraint.

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

The complete flow relation $I; C \vdash_{\mathsf{CFL}} \ell_1 \rightsquigarrow \ell_2$ defined in Figure 11 states that under constraints I and C, there is flow from label ℓ_1 to ℓ_2. The rules differ from the ones for $\mathsf{POLYFLOW}_{\mathsf{copy}}$ only in the addition of the auxiliary judgment $I; C \vdash \ell_1 \rightsquigarrow_{\mathsf{m}} \ell_2$ for *matched*

[†] The intuition applies to the first-order case.

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

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

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

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

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

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

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

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

Fig. 11. Flow relation for $\mathsf{POLYFLOW}_{\mathsf{CFL}}$

flow, and the extra rule [Match]. The sequence of instantiation edges traversed along a matched flow path form a well-parenthesised sequence, where $\preceq^i_\div$ matches $\preceq^i_+$. 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 (Henglein 1993), software dependencies (O'Callahan and Jackson 1997) and dimension inference (Rittri 1995). However, we are not aware of any previous work on 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, Melski and Reps (1997)). Given a flow graph $G = (I, C, L)$, we construct the graph G_{CFL} with nodes L, and the following labelled edges:

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

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

$$\ell_1 \xrightarrow{\mathsf{d}} \ell_2 \text{ if } \ell_1 \leqslant \ell_2 \in C\,.$$

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.

A flow relation $I ; C \vdash_{\mathsf{CFL}} \ell_1 \rightsquigarrow \ell_2$ can be derived using the rules of Figure 11 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:

$$\begin{aligned} S &\leftarrow P\ N \\ P &\leftarrow \epsilon \mid \mathsf{p} \mid \mathsf{d} \mid P\ P \mid (_i\ M\ )_i \\ N &\leftarrow \epsilon \mid \mathsf{n} \mid \mathsf{d} \mid N\ N \mid (_i\ M\ )_i \\ M &\leftarrow \epsilon \mid \mathsf{d} \mid M\ M \mid (_i\ M\ )_i\,. \end{aligned}$$

The grammar rules simply instantiate the flow rules for $p = +, \div, \mathsf{m}$. For example, the production $P \leftarrow \mathsf{d}$ is [Level] with $p = +$, the production $P \leftarrow P\ P$ is [Trans] with $p = +$, and so on. The ϵ productions arise from [Level], as $\ell \leqslant \ell$ is derivable for every label through reflexivity (Figure 3). Thus, productions for P accept exactly the paths derivable as positive flow $I ; C \vdash \ell_1 \rightsquigarrow_+ \ell_2$. The productions for N, M, and S are similarly obtained.

Since we will need later to turn the grammar into Chomsky normal form, we use an alternative, but equivalent, grammar, where we reuse the matching M derivations in the P and N productions, thereby avoiding three copies of the matching rule. The insight into this rewriting is that every M flow derivable by the rules can also be derived as either P or N flow.

$$\begin{aligned} S &\leftarrow P\ N \\ P &\leftarrow M\ P \\ &\mid\ \mathsf{p}\ P \\ &\mid\ \epsilon \\ N &\leftarrow M\ N \\ &\mid\ \mathsf{n}\ N \\ &\mid\ \epsilon \\ M &\leftarrow (_i\ M\ )_i \\ &\mid\ M\ M \\ &\mid\ \mathsf{d} \\ &\mid\ \epsilon\,. \end{aligned}$$

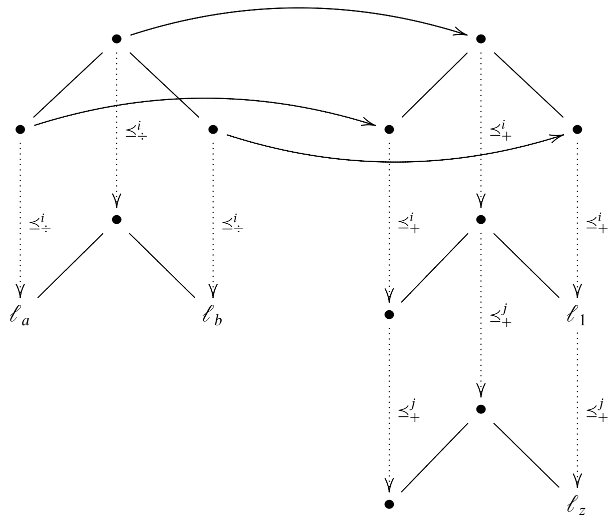
Note that in practice, the graph G_{CFL} need not be computed explicitly. Instead, the graph closure can be computed directly on the set of constraints C and I.

2.2.5. *Examples* Figure 12 shows the same example as we examined for the POLYFLOW$_{\mathsf{copy}}$ case. The upper graph is the flow graph G containing flow and instantiation edges. The lower graph is the corresponding CFL graph G_{CFL}, where we omit the d labels on flow edges. The flow path from b to z now takes the form

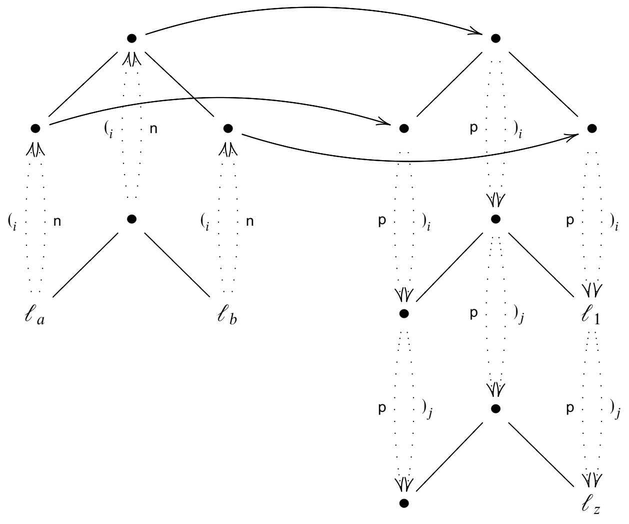
$$\ell_b \xrightarrow{(_i} \bullet \xrightarrow{\mathsf{d}} \bullet \xrightarrow{)_i} \ell_1 \xrightarrow{\mathsf{p}} \ell_z\,,$$

where the flow from ℓ_b to ℓ_1 forms a matched flow that was explicit as a copied constraint in POLYFLOW$_{\mathsf{copy}}$.

```
let idpair = λx:int × int .x in
let f      = λy:int .idpair^i (a^ℓa ,b^ℓb) in
let z      = (f^j 0).2^ℓz
...
```



Flow graph G



CFL graph G_{CFL}

Fig. 12. POLYFLOW$_{\mathsf{CFL}}$ example

```
let app = (λf.(λx. f x)^{ℓ_r})^{ℓ_app}
in
let id  = (λy.y)^{ℓ_id}
in
let w   = ((app^i id^j)^{ℓ'_r} b^{ℓ_b})^{ℓ_w}
```

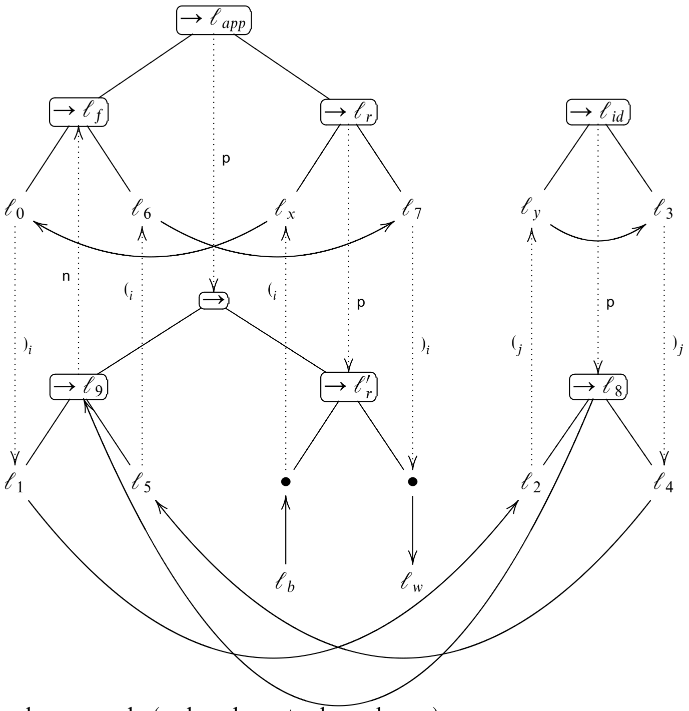


Fig. 13. Higher-order example (only relevant edges shown)

One advantage of using the type-based approach to flow analysis presented here is that it deals directly with higher-order programs. Figure 13 contains an example. The app function takes a function f as an argument and returns a function (labelled 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

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

showing that the identity function (labelled ℓ_{id}) flows to ℓ_f. The flow edges connecting the types labelled 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 $\ell_1 \leqslant \ell_2$ arises through contravariance of the subtype relation for function domains.

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

$$\ell_b \to \bullet \xrightarrow{(_i} \ell_x \to \ell_0 \xrightarrow{)_i} \ell_1 \to \ell_2 \xrightarrow{(_j} \ell_y \to \ell_3 \xrightarrow{)_j} \ell_4 \to \ell_5 \xrightarrow{(_i} \ell_6 \to \ell_7 \xrightarrow{)_i} \bullet \to \ell_w\,.$$

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 either an argument passing or a return step as in the first-order case. In this example the path traverses `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 derivation tree obtained by the rules of Figure 10 is of size 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 in practice n will typically be close to m.

Constraint derivation The type rules in Figure 10 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 5 and 3 in reverse, that is, 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 Henglein (1993) must be observed:

$$\ell \preceq^i_p \ell_1 \wedge \ell \preceq^i_{p'} \ell_2 \;\Rightarrow\; \ell_1 = \ell_2\,. \tag{F1}$$

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 $\sigma_1 \leqslant \sigma'_1$ 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. This step also obviates the need to ever apply rule (F1).

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

Fig. 14. Grammar for CFL queries.

Answering flow queries using CFL reachability After constraint derivation, we produce the graph G_{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 Melski and Reps (1997) and normalise the grammar for the CFL problem such that the right-hand sides of productions contain at most two symbols (terminals or non-terminals), which results in the grammar shown in Figure 14. This grammar 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.

Figure 15 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 $A \leftarrow B$ 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 14) 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 $\mathsf{POLYFLOW}_{\mathsf{CFL}}$.

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

The algorithm of Figure 15 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 labelled B, and the successors of ℓ reachable *via* an edge labelled C.

The algorithm assumes that given an edge labelled B, we can index through B the rules r that apply in the for loops without looking at rules that do not 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 labelled (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

```
    W = edges of $G_{\mathsf{CFL}}$
    $G_0$ = $\varnothing$
    for each production $A \leftarrow \epsilon$ 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 $A \leftarrow B$
(2)       add $\ell_1 \xrightarrow{A} \ell_2$ to $W$

        for each rule $r$ of the form $A \leftarrow B\ C$ do
          add $\ell_1$ to $pred_r(\ell_2)$
          for each $\ell_3$ in $succ_r(\ell_2)$
(3)         add $\ell_1 \xrightarrow{A} \ell_3$ to $W$
          end
        end

        for each rule $r$ of the form $A \leftarrow C\ B$ do
          add $\ell_2$ to $succ_r(\ell_1)$
          for each $\ell_0$ in $pred_r(\ell_1)$
(4)         add $\ell_0 \xrightarrow{A} \ell_2$ to $W$
          end
        end
      endif
    end
```

Fig. 15. CFL algorithm

times. Next consider the statement labelled (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 $O(en + un^2 + bn^3)$.

This bound is more precise than the bound of the algorithm presented in Melski and Reps (1997) which has worst case complexity[†] $O(|\Sigma|^3 n^3)$. 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 Melski and Reps (1997). We will take advantage of this refined complexity analysis in the next section to obtain an $O(n^3)$ algorithm.

[†] Σ is the set of terminals and non-terminals used.

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 does not 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 $O(m)$ 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 labelled $(_i$ 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 labelled 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 in the case of self-loops. Consider rule $M \leftarrow (_i\ K_i$ applied to a self-loop. The situation is

$$\begin{tikzcd} \ell \arrow[loop left, dotted, "(_i"] & \ell' \arrow[l, "K_i"'] \end{tikzcd}$$

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 $K_i \leftarrow M\ )_i$. This analysis leads to the following theorem.

Theorem 2.1. Ass ume we are given I, C, ℓ, ℓ'. Deciding whether $I; C \vdash_{\mathsf{CFL}} \ell \rightsquigarrow \ell'$ 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, which was given by Mossin (Mossin 1996), from $O(n^8)$ to $O(n^3)$. The gain is realised 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 an improved algorithm* We now show variations of the constraint derivation and the CFL-algorithm such that the initial set of constraints I that is generated is of size $O(n)$ instead of $O(mn)$. Recall that through rule [Inst], our constraint derivation phase generates up to $O(m)$ distinct instantiation self-loops of the form $\ell \xrightarrow{\preceq_p^i} \ell$ for up to $O(n)$ distinct labels ℓ annotating types of λ bound variables. Consequently, there are up to $O(n)$ nodes that need to consider rules of the form $M \leftarrow (_i\ K_i$ for up to $O(m)$ distinct i. Thus, the average number of rules that may apply per node is still $O(m)$. To

```
let g = λx:int^ℓx.
        let f = λy:int^ℓy.x
        in
          if0 x then (f^i 0^ℓ0)^ℓ2 else (f^j 1^ℓ1)^ℓ3
in
  (g^h 4^ℓ4)^ℓ5
```

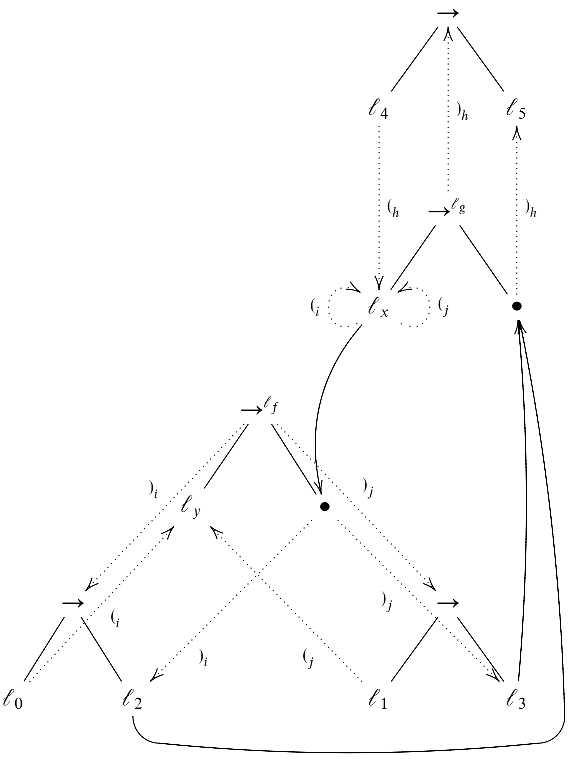


Fig. 16. Self-loop example (p and n edges not shown)

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 $\ell \preceq^i \ell$ were used in Henglein (1993) merely as a convenient way of enforcing monomorphism of λ bound variables, while in POLYFLOW$_{\mathsf{CFL}}$ self-loops are required for recovering all flow paths and thus for proving the soundness of flow.

Example Figure 16 shows an example where self-loops are present. The example contains a let binding of a function f within the scope of the λ bound parameter x of g. The body of f 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 ℓ_4 to ℓ_5, since g acts as the identity function.

The corresponding CFL-graph in Figure 16 contains two paths exhibiting the flow from ℓ_4 to ℓ_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 and the other at j. Each path uses one of the two self-loops on ℓ_x, namely the one corresponding to the particular instance of f used in the remainder of the path.

$$\ell_4 \xrightarrow{(_h} \ell_x \xrightarrow{(_i} \ell_x \rightarrow \bullet \xrightarrow{)_i} \ell_2 \rightarrow \bullet \xrightarrow{)_h} \ell_5$$

$$\ell_4 \xrightarrow{(_h} \ell_x \xrightarrow{(_j} \ell_x \rightarrow \bullet \xrightarrow{)_j} \ell_3 \rightarrow \bullet \xrightarrow{)_h} \ell_5 .$$

The self-loops are necessary to balance the result edges $\bullet \xrightarrow{)_i} \ell_2$ or $\bullet \xrightarrow{)_j} \ell_3$ 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 λ 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 λ bound variable x from an outer scope and a use of f:

```
λx : σ. ...
  ... let f = ...x...
      in
    ...    f^i
    ...
```

The self-loops on labels ℓ in σ annotating the type of a λ 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 ℓ 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 ℓ acquires a self-loop for instance i. In fact, it is sufficient to determine the relative syntactic level of λ, 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.

```
letrec f_k = λx_{k+1}. ... f_k^i ...
in
   λy_{k+1}. ... f_k^j ...
```

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

```
λx_k .
    let f_k = ...
    in
       ... f_k^i ...
```

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

The new rules are shown in Figure 17. Judgments no longer contain the sequence of free labels t, but are now indexed by the syntactic level $k : I ; C ; A \vdash_{\mathsf{CFL}_k} e : \sigma$. 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 $\sigma \preceq_+^{i,q} \sigma'$ to be annotated with the binding level q.

Level-based queries In the CFL-graph G_{CFL}, we label the instantiation edges with $(_i^q$ or $)_i^q$. Furthermore, for each label ℓ annotating the type of a lambda bound variable at level k, we add a single summary self-loop $\ell \xrightarrow{\mathsf{s}^k} \ell$ to G_{CFL}[†]. G_{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 ℓ replaces the set of self-loops with labels $(_i^q$ and $)_i^q$ where i is an instance of a binding of level $q \geqslant k$ occurring in the scope of the lambda generating $\ell \xrightarrow{\mathsf{s}^k} \ell$, we need a family of grammar rules of the form

$$M \leftarrow \mathsf{s}^k \; M \;)_i^q \qquad q \geqslant k$$

and

$$M \leftarrow (_i^q \; M \; \mathsf{s}^k \qquad q \geqslant k\,,$$

which are normalised by introducing the families of auxiliary non-terminals L^q and R^q:

$$\begin{array}{rcll} R^q & \leftarrow & M \;)_i^q & \\ L^q & \leftarrow & (_i^q \; M & \\ M & \leftarrow & \mathsf{s}^k \; R^q & q \geqslant k \\ & | & L^q \; \mathsf{s}^k & q \geqslant k\,. \end{array}$$

The full grammar is shown in Figure 18. Note that we still have no more than $O(m)$ terminals and non-terminals since the binding level q of a terminal $(_i^q$ or $)_i^q$ is determined uniquely by the instance i. However, we draw our productions from a family of m^2 distinct productions.

[†] 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.

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

Base rules

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

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

$$\frac{\vdash \tau\langle s\rangle : \sigma \qquad I;C;A,x:\sigma \vdash_{\mathsf{CFL}_k} e:\sigma'}{I,C;A \vdash_{\mathsf{CFL}_k} \lambda^\ell x{:}\tau.e : \sigma \rightarrow^\ell \sigma'}\,[\text{Lam}]$$

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

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

$$\frac{I;C;A \vdash_{\mathsf{CFL}_k} e_0 : int^\ell \qquad I;C;A \vdash_{\mathsf{CFL}_k} e_1 : \sigma \qquad I;C;A \vdash_{\mathsf{CFL}_k} e_2 : \sigma}{I;C;A \vdash_{\mathsf{CFL}_k} \texttt{if0}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \sigma}\,[\text{Cond}]$$

$$\frac{I;C;A \vdash_{\mathsf{CFL}_k} e:\sigma \qquad C \vdash \sigma \leqslant \sigma'}{I;C;A \vdash_{\mathsf{CFL}_k} e:\sigma'}\,[\text{Sub}] \qquad \frac{\begin{array}{c} I;C;A \vdash_{\mathsf{CFL}_k} e:\sigma \\ \vdash \tau\langle \ell' s\rangle : \sigma \qquad C \vdash \ell = \ell' \end{array}}{I;C;A \vdash_{\mathsf{CFL}_k} e^\ell : \sigma}\,[\text{Label}]$$

Polymorphic rules

$$\frac{\begin{array}{c} I;C;A \vdash_{\mathsf{CFL}_{k+1}} e_1 : \sigma_1 \qquad \vec{\ell} = gen(A,\sigma_1) \\ I;C;A,f:(\forall\vec{\ell}.\sigma_1,k) \vdash_{\mathsf{CFL}_{k+1}} e_2 : \sigma_2 \end{array}}{I;C;A \vdash_{\mathsf{CFL}_k} \texttt{let}\ f\ \texttt{=}\ e_1\ \texttt{in}\ e_2 : \sigma_2}\,[\text{Let}]$$

$$\frac{\begin{array}{c} I;C;A,f:(\forall\vec{\ell}.\sigma_1,k) \vdash_{\mathsf{CFL}_{k+1}} e_1 : \sigma_1 \qquad \vec{\ell} = gen(A,\sigma_1) \\ I;C;A,f:(\forall\vec{\ell}.\sigma_1,k) \vdash_{\mathsf{CFL}_{k+1}} e_2 : \sigma_2 \end{array}}{I;C;A \vdash_{\mathsf{CFL}_k} \texttt{letrec}\ f\ \texttt{=}\ e_1\ \texttt{in}\ e_2 : \sigma_2}\,[\text{Rec}]$$

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

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

Fig. 17. $\mathsf{POLYFLOW}_{\mathsf{CFL}}$ with levels

We now have on average a constant number of edges labelled with terminals incident on each node in our graph, and the average number of distinct rules that apply per node is constant, except for nodes ℓ with summary self-loops s^k, since they need to consider the family of rules $M \leftarrow \mathsf{s}^k\ R^q$ (and $M \leftarrow L^q\ \mathsf{s}^k$) for all q such that $q \geqslant k$. However, it is still the case that for a fixed ℓ and q, the number of applications of this rule is bounded by $O(n)$ since at most n distinct labels ℓ' are connected to ℓ *via* an edge labelled R^q. Even though we have reduced the number of initial edges to $O(n)$, the CFL-algorithm

$$
\begin{array}{rcll}
S & \leftarrow & P\ N & \\
P & \leftarrow & M\ P & \\
 & | & \mathsf{p}\ P & \\
 & | & \epsilon & \\
N & \leftarrow & M\ N & \\
 & | & \mathsf{n}\ N & \\
 & | & \epsilon & \\
K_i & \leftarrow & M\ )_i^q & \\
M & \leftarrow & (_i^q\ K_i & \\
 & | & M\ M & \\
 & | & \mathsf{d} & \\
 & | & \epsilon & \\
 & | & \mathsf{s}^k\ R^q & q \geqslant k \\
 & | & L^q\ \mathsf{s}^k & q \geqslant k \\
R^q & \leftarrow & M\ )_i^q & \\
L^q & \leftarrow & (_i^q\ M &
\end{array}
$$

Fig. 18. Grammar for summary self-loops

of Figure 15 would require $O(m)$ sets $succ_r$ at a label ℓ with summary self-loop k, one for each rule with $q \geqslant k$. Overall, that will lead to $O(mn)$ such sets. We can modify the algorithm to use only $O(n)$ such sets by noting that at any particular label ℓ where the family of rules $M \leftarrow \mathsf{s}^k\ R^q$ with $q \geqslant k$ applies, the set $succ_r$ of all these rules can be shared because it does not matter which particular level q is used to trigger the rule. This optimisation 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 $M \leftarrow L^q\ \mathsf{s}^k$.

To summarise, the introduction of summary self-loops reduces the size of the initial constraint graph from $O(mn)$ to $O(n)$, and, furthermore, allows the CFL-algorithm to use only constant number of $pred_r$ and $succ_r$ sets per node on average.

Example Figure 19 shows the same example as Figure 16, but with summary self-loops and levels. The bindings for f, g, x and y are subscripted with their level. Consider again the flow from ℓ_4 to ℓ_5. There are two paths, corresponding to the two branches of the conditional:

$$\ell_4 \xrightarrow{(_{h,0}} \ell_x \xrightarrow{\mathsf{s}^1} \ell_x \rightarrow \bullet \xrightarrow{)_{i,1}} \ell_2 \rightarrow \bullet \xrightarrow{)_{h,0}} \ell_5$$

$$\ell_4 \xrightarrow{(_{h,0}} \ell_x \xrightarrow{\mathsf{s}^1} \ell_x \rightarrow \bullet \xrightarrow{)_{j,1}} \ell_3 \rightarrow \bullet \xrightarrow{)_{h,0}} \ell_5$$

Both paths combine the self-loop s^1 on x with a closing parenthesis. The flow between ℓ_x and ℓ_2 (respectively, ℓ_3) is computed using the following production:

$$M \leftarrow \mathsf{s}^k\ M\ )_i^q \qquad q \geqslant k\,.$$

The production applies, as we have $q = 1$, which is the binding occurrence of f reflected in the level annotation on the closing parenthesis, as well as $k = 1$, which is the level of lambda bound x.

```
let g$_0$ = $\lambda$x$_1$:$int^{\ell_x}$.
        let f$_1$ = $\lambda$y$_2$:$int^{\ell_y}$.x
        in
          if0 x then (f$^i$ 0$^{\ell_0}$)$^{\ell_2}$ else (f$^j$ 1$^{\ell_1}$)$^{\ell_3}$
in
  (g$^h$ 4$^{\ell_4}$)$^{\ell_5}$
```

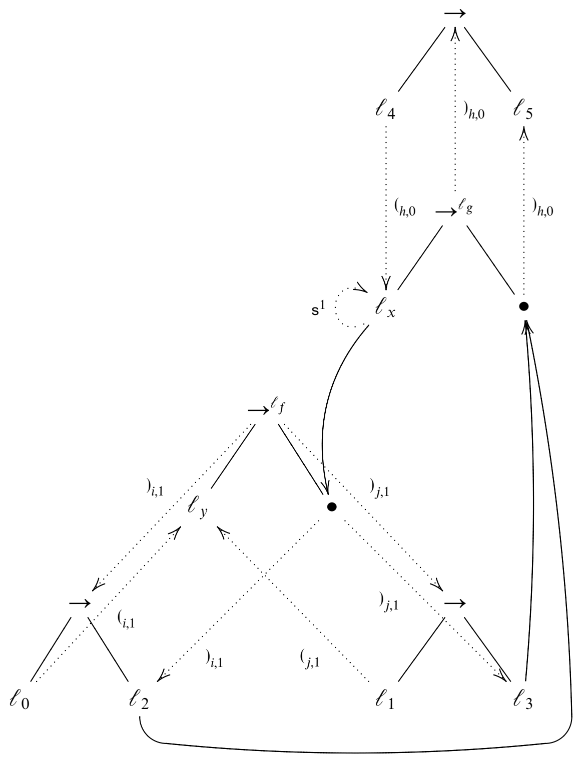


Fig. 19. Self-loops with levels (p and n edges not shown)

Levels instead of scopes By construction, the set of flow paths recognised 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 ℓ_1 on a lambda bound variable y at level p may match up with any instantiation edge $\ell_2 \xrightarrow{)_i^q} \ell_3$ with a binding level q greater than or equal to p, even though the instance i does not appear in the scope of λy (see Figure 20). But for such a situation to arise, there must be an edge $\ell_1 \xrightarrow{R^q} \ell_3$ that, in turn, requires a matched path connecting label ℓ_1 with the instantiation edge $)_i^q$ at label ℓ_2. Such a matched path must go through an outer lambda λx at level $k \leqslant p$ with label ℓ_0, where λx contains both λy and the instantiation i in its

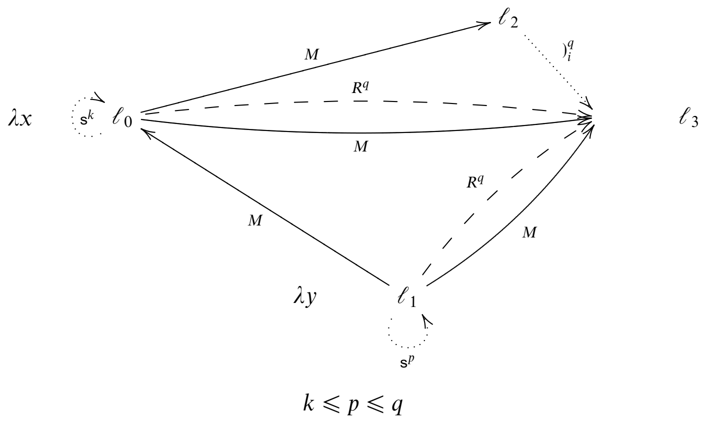


Fig. 20. Matching self-loops with instantiations of different scopes

scope (refer to Figure 20). The label ℓ_0 will have a self-loop that can properly match up with the instantiation edge $)_i^q$, thereby producing a direct edge $\ell_0 \xrightarrow{M} \ell_3$. Thus, by transitivity of M-edges, the matched path from ℓ_1 to ℓ_0 combined with the path from ℓ_0 to ℓ_3 gives rise to the edge $\ell_1 \xrightarrow{M} \ell_3$, without the need for the self-loop at ℓ_1.

Thus, even though the edge from ℓ_1 to ℓ_3 in Figure 20 may erroneously be added by applying the rule $M \leftarrow \mathsf{s}^p\ R^q$, 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 (Pessaux 2000, Chapter 8). See also the chapter by Pottier and Rémy in Pottier and Rémy (2005).

2.2.8. *Demand-driven algorithm* We now sketch a demand-driven CFL-closure algorithm using the ideas of Reps *et al.* (1995). The grammar of Figure 18 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 Reps *et al.* (1995). Figure 21 shows the resulting grammar for a backward demand algorithm (which requires a modification of the rules involving L^q). Given a label ℓ, we ask for all labels that can potentially flow to ℓ. We seed the algorithm with a trigger edge $\ell \xrightarrow{T} \ell$ and an edge $\ell \xrightarrow{N} \ell$. We also use three extra rules in the CFL algorithm:

Given	Add	
$\ell_1 \xrightarrow{)_i} \bullet \xrightarrow{T} \bullet$	$\ell_1 \xrightarrow{T} \ell_1$	$D1$
$\ell_1 \xrightarrow{\mathsf{s}^k} \bullet \xrightarrow{T} \bullet$	$\ell_1 \xrightarrow{T} \ell_1$	$D3$
$\ell_1 \xrightarrow{N} \bullet$	$\ell_1 \xrightarrow{P} \ell_1$	$D3$

$$
\begin{array}{rcll}
S & \leftarrow & P\ N & \\
P & \leftarrow & T\ P & \\
 & | & \mathsf{p}\ P & \\
N & \leftarrow & T\ N & \\
 & | & \mathsf{n}\ N & \\
K_i & \leftarrow & T\ )_i^q & \\
M & \leftarrow & (_i^q\ K_i & \\
 & | & \mathsf{d} & \\
 & | & \mathsf{s}^k\ R^q & q \geqslant k \\
 & | & (_i^q\ L^k & q \geqslant k \\
R^q & \leftarrow & T\ )_i^q & \\
L^k & \leftarrow & T\ \mathsf{s}^k & \\
T & \leftarrow & M\ T &
\end{array}
$$

Fig. 21. Grammar for demand-driven algorithm

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 ℓ' connected to ℓ by an S-edge.

2.3. *Soundness*

Soundness of our flow relation $I\,; C \vdash_{\mathsf{CFL}} \ell_1 \rightsquigarrow \ell_2$ 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 of showing that for every derivation in $\mathsf{POLYFLOW}_{\mathsf{CFL}}$, there exists a derivation in $\mathsf{POLYFLOW}_{\mathsf{copy}}$ such that our notion of flow is a safe approximation to the flow defined by the copy-based system (in fact, our notion of flow is equivalent, in terms of precision). The system $\mathsf{POLYFLOW}_{\mathsf{copy}}$ is similar to a standard copy-based system studied by Mossin for which soundness has been established (Mossin 1996)).

A technical core in our proof consists of 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 (Mossin 1996), soundness of our notion of flow follows.

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

Theorem 2.2. (Soundness) For every judgment

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

derivable in $\mathsf{POLYFLOW}_{\mathsf{CFL}}$, there exists a judgment

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

(Expressions)	e	$::=$	$x \mid n \mid$ `(`e_1`,` e_2`)` $\mid \lambda x{:}\tau.e \mid e_1\, e_2 \mid$
			`let` $f : \forall\vec{\alpha}.\tau$ `=` e_1 `in` $e_2 \mid f^i_\tau \mid$
			`letrec` $f : \forall\vec{\alpha}.\tau$ `=` e_1 `in` $e_2 \mid$
			`if0` e_0 `then` e_1 `else` $e_2 \mid e.i$
(Types)	τ	$::=$	$int \mid \tau \rightarrow \tau \mid \tau \times \tau \mid \alpha$
(Labelled Types)	$\tau\langle s\rangle, \sigma$	$::=$	$int^\ell \mid \sigma \rightarrow^\ell \sigma \mid \sigma \times^\ell \sigma \mid \alpha^h$
(Labels)	L	$::=$	$\ell \mid h$
(Label Sequence)	s, t	$::=$	$\ell \mid h^\alpha \mid s\, s$

Fig. 22. Definitions

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

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

3. Flow and type-structure polymorphism

In this chapter we extend the flow analysis of Section 2 to include polymorphism over type structure. In Section 3.1 we show how to extend the CFL-based approach of POLYFLOW$_{\mathsf{CFL}}$ to encompass type-structure polymorphism. We then show in Section 3.2 how to specialise the approach to a flow-analysis system based on purely equational reasoning (rather than subtyping), thereby showing how the technique described in Fähndrich *et al.* (2000) arises from the general CFL-based approach of Rehof and Fähndrich (2001) in a systematic way, by specialisation.

We extend the definitions from Figure 1 as shown in Figure 22. The language of types now contains type variables α. Type annotation on let and letrec bindings take the form $\forall\vec{\alpha}.\tau$. Labelled types σ are extended to type variables, where occurrences of type variables are annotated with a special kind of label h called a *hole label.* We require that a hole label annotates at most one type variable. Label sequences thus contain both ordinary labels ℓ and hole labels h^α, where the hole labels are tagged with the type variable that they annotate in the type.

The well-labelling logic of Figure 2 is extended with the rule

$$\frac{}{\vdash \alpha\langle h^\alpha\rangle : \alpha^h},$$

and, similarly, the subtype relation from Figure 4 is extended with the following rule:

$$\frac{C \vdash h_1 \leqslant h_2}{C \vdash^{\leqslant} \alpha^{h_1} \leqslant \alpha^{h_2}}\,[\text{Var}]$$

In the same vein, we extend the instantiation relation of Figure 5 to type variables:

$$\frac{I \vdash h \preceq^i_p s \qquad I \vdash \alpha \preceq^i_p \tau}{I \vdash \alpha^h \preceq^i_p \tau\langle s\rangle}\,[\text{Var}]$$

Note that we now have instantiation constraints on type variables of the form $\alpha \preceq^i_p \tau$.

Substitutions φ now have three components:

— $\varphi|_\alpha$ maps type variables α to unlabelled types τ;

— $\varphi|_\ell$ maps ordinary labels ℓ to ordinary labels ℓ';
— $\varphi|_h$ maps hole labels to label sequences s.

The domain of a substitution φ is the set of type variables and labels on which φ is not the identity. We say that a substitution φ is *well formed* for a particular type $\sigma = \tau\langle s\rangle$ if the result σ' of the substitution is well labelled, that is, $\vdash \varphi(\tau)\langle\varphi(s)\rangle : \sigma'$. In other words, for all $\alpha \in \tau$ and for all $h^\alpha \in s$, we have $\vdash \varphi(\alpha)\langle\varphi(h^\alpha)\rangle :$ wl, that is, the structure substitution of a type variable α and the sequence substitutions of all hole labels tagged with α yield well-labelled types.

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

$$int^{\ell_2} \times^{\ell_1} int^{\ell_3} \rightarrow^{\ell_0} int^{\ell_5} \times^{\ell_4} int^{\ell_6}\,.$$

3.1. *CFL-based approach*

We will skip the copy-based approach and show directly how to extend the CFL-based approach of POLYFLOW$_{\mathsf{CFL}}$ to POLYTYPE$_{\mathsf{CFL}}$.

Polymorphic types Polymorphic types are now written $\forall\alpha_i\ell_i h_i.\tau\langle s\rangle$, with quantified variables α_i and quantified labels ℓ_i, h_i. A quantified type is said to be well labelled (written $\vdash \forall\alpha_i\ell_i h_i.\tau\langle s\rangle$ wl), if for all α_i, all hole variables $h_k^\alpha \in s$ are among the quantified labels h_i. In other words, hole labels annotating occurrences of quantified variables α must themselves be quantified. We use $ftv(\sigma)$ to denote the free type variables of σ and extend this to type environments A in the standard way.

A type $\tau'\langle s'\rangle$ is an *instance* of a polymorphic type $\forall\vec{\alpha}\vec{\ell}\vec{h}.\tau\langle s\rangle$, written $\forall\vec{\alpha}\vec{\ell}\vec{h}.\tau\langle s\rangle \preceq \tau'\langle s'\rangle$, if there exists a substitution φ over the quantified variables such that $\varphi(\tau) = \tau'$, $\varphi(s) = s'$ and $\vdash \tau'\langle s'\rangle$ wl.

The set of free hole labels of types and environments are written $fh(\sigma)$ and $fh(A)$, respectively.

3.1.1. *Type rules*

Figure 23 shows the rules for the type polymorphic flow system. The base rules are identical to the rules for POLYFLOW$_{\mathsf{CFL}}$ in Figure 17. 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 polymorphic types in the letrec case are given through some other means (by the programmer or a semi-decidable procedure (Henglein 1993)).

For completeness, the type rules are presented with the level optimisation and summary self-loops discussed in Section 2.2.7. In the rest 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.

$$\boxed{I\,;C\,;A \vdash_k^{pt} e : \sigma}$$

Base rules

$$\frac{}{I\,;C\,;A, x : \sigma \vdash_k^{pt} x : \sigma}\,[\mathrm{Id}] \qquad \frac{}{I\,;C\,;A \vdash_k^{pt} n^\ell : int^\ell}\,[\mathrm{Int}]$$

$$\frac{I\,;C\,;A \vdash_k^{pt} e_1 : \sigma_2 \to^\ell \sigma_1 \qquad I\,;C\,;A \vdash_k^{pt} e_2 : \sigma_2}{I\,;C\,;A \vdash_k^{pt} e_1\ e_2 : \sigma_1}\,[\mathrm{App}]$$

$$\frac{\vdash \tau\langle s\rangle : \sigma \qquad I\,;C\,;A, x : \sigma \vdash_k^{pt} e : \sigma'}{I, C\,;A \vdash_k^{pt} \lambda^\ell x{:}\tau.e : \sigma \to^\ell \sigma'}\,[\mathrm{Lam}]$$

$$\frac{I\,;C\,;A \vdash_k^{pt} e_1 : \sigma_1 \qquad I\,;C\,;A \vdash_k^{pt} e_2 : \sigma_2}{I\,;C\,;A \vdash_k^{pt} (e_1, e_2)^\ell : \sigma_1 \times^\ell \sigma_2}\,[\mathrm{Pair}]$$

$$\frac{I\,;C\,;A \vdash_k^{pt} e : \sigma_1 \times^\ell \sigma_2}{I\,;C\,;A \vdash_k^{pt} e.i : \sigma_i}\,[\mathrm{Proj}\ i = 1, 2]$$

$$\frac{I\,;C\,;A \vdash_k^{pt} e_0 : int^\ell \qquad I\,;C\,;A \vdash_k^{pt} e_1 : \sigma \qquad I\,;C\,;A \vdash_k^{pt} e_2 : \sigma}{I\,;C\,;A \vdash_k^{pt} \texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \sigma}\,[\mathrm{Cond}]$$

$$\frac{I\,;C\,;A \vdash_k^{pt} e : \sigma \qquad C \vdash \sigma \leqslant \sigma'}{I\,;C\,;A \vdash_k^{pt} e : \sigma'}\,[\mathrm{Sub}] \qquad \frac{\begin{array}{c} I\,;C\,;A \vdash_k^{pt} e : \sigma \\ \vdash \tau\langle \ell' s\rangle : \sigma \qquad C \vdash \ell = \ell' \end{array}}{I\,;C\,;A \vdash_k^{pt} e^\ell : \sigma}\,[\mathrm{Label}]$$

Polymorphic rules

$$\frac{\begin{array}{l} I\,;C\,;A \vdash_{k+1}^{pt} e_1 : \sigma_1 \\ I\,;C\,;A, f : (\forall \vec{\alpha}\vec{\ell}\vec{h}.\sigma_1, k) \vdash_{k+1}^{pt} e_2 : \sigma_2 \\ \vec{\alpha} = gen(A, \sigma_1) \qquad \vec{\ell} = gen_\ell(A, \sigma_1) \qquad \vec{h} = gen_h(A, \sigma_1) \\ \vdash \tau\langle s\rangle : \sigma_1 \qquad \vdash \forall \vec{\alpha}\vec{\ell}\vec{h}.\sigma_1\ \mathsf{wl} \end{array}}{I\,;C\,;A \vdash_k^{pt} \texttt{let}\ f : \forall\vec{\alpha}.\tau\ \texttt{=}\ e_1\ \texttt{in}\ e_2 : \sigma_2}\,[\mathrm{Let}]$$

$$\frac{\begin{array}{l} I\,;C\,;A, f : (\forall \vec{\alpha}\vec{\ell}\vec{h}.\sigma_1, k) \vdash_{k+1}^{pt} e_1 : \sigma_1 \\ I\,;C\,;A, f : (\forall \vec{\alpha}\vec{\ell}\vec{h}.\sigma_1, k) \vdash_{k+1}^{pt} e_2 : \sigma_2 \\ \vec{\alpha} = gen(A, \sigma_1) \qquad \vec{\ell} = gen_\ell(A, \sigma_1) \qquad \vec{h} = gen_h(A, \sigma_1) \\ \vdash \tau\langle s\rangle : \sigma_1 \qquad \vdash \forall \vec{\alpha}\vec{\ell}\vec{h}.\sigma_1\ \mathsf{wl} \end{array}}{I\,;C\,;A \vdash_k^{pt} \texttt{letrec}\ f : \forall\vec{\alpha}.\tau\ \texttt{=}\ e_1\ \texttt{in}\ e_2 : \sigma_2}\,[\mathrm{Rec}]$$

$$\frac{I \vdash \sigma \preceq_+^{i,q} \sigma' : \varphi \qquad dom\,\varphi = \vec{\alpha}, \vec{\ell}, \vec{h}}{I\,;C\,;A, f : (\forall \vec{\alpha}\vec{\ell}\vec{h}.\sigma, q) \vdash_k^{pt} f^i : \sigma'}\,[\mathrm{Inst}]$$

$gen(A, \sigma) = ftv(\sigma) \setminus ftv(A)$
$gen_\ell(A, \sigma) = fl(\sigma) \setminus fl(A)$
$gen_h(A, \sigma) = fh(\sigma) \setminus fh(A)$

Fig. 23. POLYTYPE$_{\mathsf{CFL}}$

```
let id = λx:α. x in
let f  = λy:β. id^i (a^ℓa, id^k y^ℓy)^ℓp in
let z  = (f^j b^ℓb).2^ℓz
...
```

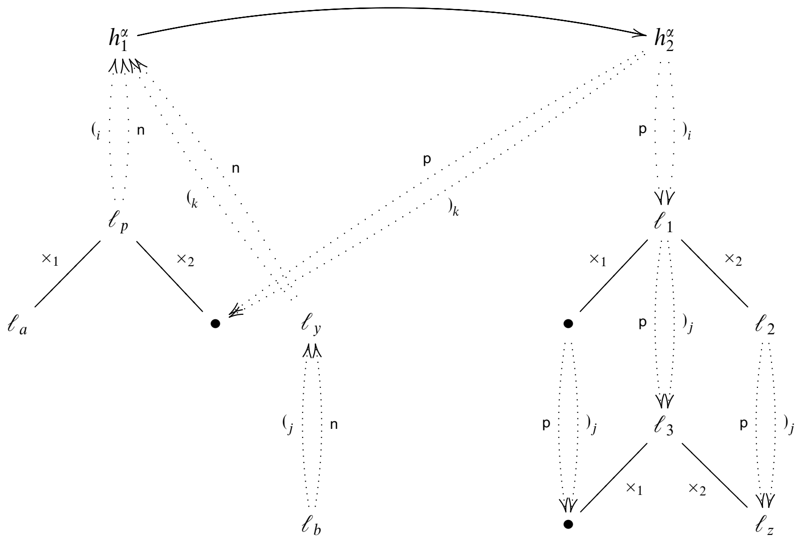


Fig. 24. Type-polymorphic example

3.1.2. *Flow queries in the presence of type polymorphism* Consider the example of Figure 24, which involves the application of the polymorphic identity function `id` with type $\forall\alpha h_1 h_2.\alpha^{h_1} \rightarrow \alpha^{h_2}$. We are again interested in the flow from b to z, but, whereas in earlier examples (for example, Figure 12) there existed a CFL path from ℓ_b to ℓ_z involving only instantiation and flow constraints, the flow paths in the type polymorphic case of Figure 24 involve traversing constructor edges, that is, edges linking the pair type constructors with its right sub-component. We view constructor edges as a pair of directed edges, one from parent to child, and the other from child to parent. The child to parent edge is labelled with an open parenthesis indexed by the parent type constructor and the child index. For example, the edge from ℓ_a to the pair type labelled ℓ_p is labelled $[_{\times_1}$. Similarly, parent to child edges are labelled with closing parenthesis, in our example $]_{\times_1}$. To avoid clutter in our graphs, we do not draw both edges and simply annotate the undirected edge with the constructor/index label ($\times_1$ in our example).

The flow from b to z is shown by the following path, which is accepted by the grammar in Figure 25:

$$\ell_b \underbrace{\xrightarrow{(_j} \ell_y \underbrace{\xrightarrow{(_k} h_1 \rightarrow h_2 \xrightarrow{)_k}} \bullet \underbrace{\xrightarrow{[_{\times_2}} \ell_p \underbrace{\xrightarrow{(_i} h_1 \rightarrow h_2 \xrightarrow{)_i}} \ell_1 \xrightarrow{]_{\times_2}}} \ell_2 \xrightarrow{)_j}} \ell_z$$

There are only two new productions with respect to the grammar for $\mathsf{POLYFLOW}_{\mathsf{CFL}}$, namely $M \leftarrow [_{c_i}\ M\ ]_{c_i}$ and $M \leftarrow [_{c_i}\ \overline{M}\ ]_{c_i}$. The first new production enables us to match

$$
\begin{array}{rcll}
S & \leftarrow & P\ N & \\
P & \leftarrow & M\ P & \\
 & | & \mathsf{p}\ P & \\
 & | & \epsilon & \\
N & \leftarrow & M\ N & \\
 & | & \mathsf{n}\ N & \\
 & | & \epsilon & \\
M & \leftarrow & (_i\ M\ )_i & \\
 & | & [_{c_i}\ M\ ]_{c_i} & c_i \text{ covariant in } c \\
 & | & [_{c_i}\ \overline{M}\ ]_{c_i} & c_i \text{ contravariant in } c \\
 & | & M\ M & \\
 & | & \mathsf{d} & \\
 & | & \epsilon &
\end{array}
$$

Fig. 25. Extended grammar for constructor matching

edges from the ith child of a type constructor c and 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

$$\text{if } \ell_1 \xrightarrow{M} \ell_2 \text{ is an edge in } G_0, \text{ then } \ell_2 \xrightarrow{\overline{M}} \ell_1 \text{ is also an edge in } G_0. \qquad \text{(RevM)}$$

We could express reversed M-paths directly in the grammar, but we would have to introduce reverse edges for all terminals involved in M productions and write corresponding productions for $\overline{M}$ as follows:

$$
\begin{array}{rcl}
\overline{M} & \leftarrow & \overline{)_i}\ \overline{M}\ \overline{(_i} \\
 & | & \overline{M}\ \overline{M} \\
 & | & \overline{\mathsf{d}}\,.
\end{array}
$$

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 (Reps 2000). 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 24 formed by

$$\ell_b \xrightarrow{(_j} \ell_y \xrightarrow{(_k} h_1 \rightarrow h_2 \xrightarrow{)_k} \bullet \xrightarrow{[_{\times_2}} \ell_p \xrightarrow{(_i} h_1 \rightarrow h_2 \xrightarrow{)_i} \ell_1 \xrightarrow{)_j} \ell_3 \xrightarrow{]_{\times_2}} \ell_z\,.$$

This path differs from our earlier path in the last two edges traversed. Instead of traversing the edge $\ell_1 \xrightarrow{]_{\times_2}} \ell_2$ followed by $\ell_2 \xrightarrow{)_j} z$, we take $\ell_1 \xrightarrow{)_j} \ell_3$ followed by $\ell_3 \xrightarrow{]_{\times_2}} z$. The second path matches what happens operationally, namely, that we return the pair from function `f` along the edge $\ell_1 \xrightarrow{)_j} \ell_3$ and then select the second component of the pair. However, on this path, the constructor and instantiation parentheses are not properly nested, since $[_{\times_2}$ does not match up with $)_j$. If we interpret the first flow path operationally, we see that we

$$\begin{tikzcd}
& \ell_2 \arrow[rr, dotted, "t"] & & \ell_3 \\
\ell_1 \arrow[ur, dash, "c_i"] \arrow[rr, dotted, "t"] & & \ell'_2 \arrow[ur, dash, "c_i"] &
\end{tikzcd}$$

Fig. 26. Commuting path

select the second component of the pair within function `f` prior to returning it in order to properly nest $[_{\times_2}$ and $]_{\times_2}$ within $(_j$ and $)_j$.

The reason we can always achieve proper nesting is that we work with a *structural* subtyping system, where subsumption steps $\sigma_1 \leqslant \sigma_2$ always require that the type structure of σ_1 and σ_2 are the same. As a result, the type structure contains enough explicit flow edges that without type polymorphism (that is, $\mathsf{POLYFLOW_{CFL}}$) there is no need to traverse constructor edges at all, even though the type language contains them. With type polymorphism, the only time a flow path must traverse a constructor edge is when the flow path enters a type polymorphic function and the type structure in the polymorphic function is not explicit enough to support the flow path. The intuition for the existence of properly nested flow paths is thus that constructor edges need only occur surrounding applications of type-polymorphic functions, and therefore with proper nesting.

Theorem 3.1 (Properly nested flow paths). Given a path from ℓ_s to ℓ_t where

1 the projection of constructor brackets is well-balanced and
2 the projection of call–return parentheses is a PN path,

there exists an alternative flow-path from ℓ_s to ℓ_t reducing to non-terminal S in the grammar of Figure 25 (that is, where constructor and call–return parentheses are properly nested).

Proof. We explicitly construct an alternate path from ℓ_s to ℓ_t that is properly nested by commuting opening brackets forward and closing brackets backwards until they either annihilate or surround a polymorphic type instantiation.

Observe that any subpath of the form $\ell_1 \xrightarrow{[_{c_i}} \ell_2 \xrightarrow{t} \ell_3$, where t is either d, $(_j$, or $)_j$ and ℓ_3 is not a hole label, exhibits an alternate path from ℓ_1 to ℓ_3 that commutes the edge labels. The type at ℓ_2 is clearly of the form $c(...)$. Then, by rule [Sub] or [Inst], the type at ℓ_3 is also of the form $c(.,\ell'_2,..)$ (where ℓ'_2 is the ith argument to c). Furthermore, by these type rules, there is also an edge $\ell_1 \xrightarrow{t} \ell'_2$ (see Figure 26). Thus, the alternate path is $\ell_1 \xrightarrow{t} \ell'_2 \xrightarrow{[_{c_i}} \ell_3$. A similar argument applies for changing subpaths $\ell_1 \xrightarrow{t} \ell_2 \xrightarrow{]_{c_i}} \ell_3$ into $\ell_1 \xrightarrow{]_{c_i}} \ell'_2 \xrightarrow{t} \ell_3$.

Any cycle of the form $\ell_1 \xrightarrow{[_{c_i}} \ell_2 \xrightarrow{]_{c_i}} \ell_1$ can be removed without changing the flow. Note that the alternate paths constructed as above maintain the balanced parentheses property for both constructor and call–return parentheses since we only commute parentheses belonging to distinct groups.

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

$$\frac{\begin{array}{cc} c^{\ell_1}(..,\tau_i\langle \ell_2 s\rangle,..) \in D & c^{\ell_3}(..,\tau_i'\langle \ell_4 s'\rangle,..) \in D \\ c \text{ is co-variant in } i & I\,; C \vdash \ell_1 \rightsquigarrow_{\mathsf{m}} \ell_3 \end{array}}{I\,; C \vdash \ell_2 \rightsquigarrow_p \ell_4}\,[\text{Match(Cov)}]$$

$$\frac{\begin{array}{cc} c^{\ell_1}(..,\tau_i\langle \ell_2 s\rangle,..) \in D & c^{\ell_3}(..,\tau_i'\langle \ell_4 s'\rangle,..) \in D \\ c \text{ is contra-variant in } i & I\,; C \vdash \ell_3 \rightsquigarrow_{\mathsf{m}} \ell_1 \end{array}}{I\,; C \vdash \ell_2 \rightsquigarrow_p \ell_4}\,[\text{Match(Contra)}]$$

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

Fig. 27. Extended flow relation for $\mathsf{POLYTYPE}_{\mathsf{CFL}}$

After applying the above rules to exhaustion, we obtain a path where all opening brackets appear in sequences ending in an open parenthesis leading to a hole variable: $\ell_1 \xrightarrow{[} \cdots \xrightarrow{[} \ell_n \xrightarrow{(_i} h^\alpha$ and, similarly, closing brackets appear after a closing parenthesis $h' \xrightarrow{)_i} \ell_1' \xrightarrow{]} \cdots \xrightarrow{]} \ell_m'$.

Now consider a pair of such sequences where the parenthesis $(_i$ and $)_i$ form a matching pair for the path. We can argue that the immediately surrounding brackets also form a matching pair because the type at h^α and h' is abstract (type variable α). It is therefore not possible that the first closing bracket] matches anywhere on the path $h \rightsquigarrow h'$, and as a result, it must match the immediately preceeding opening bracket.

Next, note that m and n might not be equal, and hence, either the opening sequence has an unmatched prefix ($n > m$) or the closing sequence has an unmatched suffix ($n < m$). In the former case, let s be the unmatched prefix and ℓ_j be the label separating the prefix. We can prolong the path on the closing sequence with a loop path in the type structure rooted at ℓ_m' spelling out $\bar{s}s$ (where $\bar{s}$ is s reversed). This path exists, because the type at ℓ_m' is equal to the type at ℓ_j. Furthermore, this path lengthening still keeps the overall brackets balanced. We can perform a similar adjustment in the case $n < m$.

Finally, we repeat the propagation of opening brackets forward and closing brackets backwards followed by the adjustments until all sequences are balanced. This process is finite, as the type structure is finite. When all bracket sequences are balanced, we have obtained a properly nested sequence of both call–return parentheses and constructor brackets, as all brackets appear immediately surrounding a matched call–return parenthesis pair and the sequences match up completely. □

Figure 27 shows the flow relation extended to type constructors. We use the notation $c^\ell(..,\tau_i\langle \ell' s\rangle,..) \in D$ to mean that the type $c^\ell(..,\tau_i\langle \ell' s\rangle,..)$ with label ℓ appears somewhere in the type derivation (D) and that the ith constructor argument is type $\tau_i\langle \ell' s\rangle$, labelled with ℓ'.

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

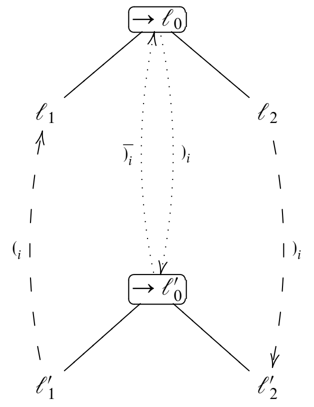


Fig. 28. Graphical illustration of extra productions

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

$$\begin{array}{lcll}
)_i & \leftarrow & [_{c_i}\ )_i\ ]_{c_i} & c_i \text{ covariant in } c \\
(_i & \leftarrow & [_{c_i}\ \overline{)_i}\ ]_{c_i} & c_i \text{ contravariant in } c \\
(_i & \leftarrow & [_{c_i}\ (_i\ ]_{c_i} & c_i \text{ covariant in } c \\
)_i & \leftarrow & [_{c_i}\ \overline{(_i}\ ]_{c_i} & c_i \text{ contravariant in } c\,.
\end{array}$$

Figure 28 illustrates the application of the first two rules to a function type constructor. The original instantiation edge $\ell_0 \xrightarrow{)_i} \ell'_0$ gives rise to the two edges $\ell_2 \xrightarrow{)_i} \ell'_2$ and $\ell'_1 \xrightarrow{(_i} \ell_1$.

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 unlabelled type structure can be represented as a DAG with size $O(m)$, whereas the labelled type structure is a tree of the same shape, because shared type structures are labelled 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 unlabelled type graph can itself be exponential in m, and the labelled 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. However, in practice we expect the demand-driven queries to explore only small parts of the labelled structure.

We have not 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 σ_1. This guarantees that any label in σ_1 can be instantiated to a fresh label at all instances.

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

$$\begin{aligned} T &: L \to \tau \\ M &: L \to \mathbb{N} \to L\,. \end{aligned}$$

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 labelled type structure. Given a label ℓ and a child index k, $M\ \ell\ k$ produces the label annotating the kth child position of the labelled type annotated by ℓ. Initially, the two maps are empty, and as constraint derivation proceeds, the mappings T and M are extended as 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*

Because of its good running time in practice, context-sensitivity based on Hindley–Milner style polymorphic type inference is in widespread use. 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 α has a single unique hole label h^α used at every occurrence of α.

— 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 modelled by equivalence classes. The flow of values between different instantiation contexts is characterised by instantiation edges, which are directed.

3.2.1. *Type system* As described above, the type system of POLYEQ differs from POLYTYPE only in the omission of the [Sub] rule, along with the restriction that if two hole variables h_1^α and h_2^α are associated with the same type variable α, then $h_1^\alpha = h_2^\alpha$, so we will 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 25. We reproduce the productions for non-terminal M representing matched flow below:

$$\begin{array}{rcll} M & \leftarrow & (_i\ M\ )_i & \\ & | & [_{c_i}\ M\ ]_{c_i} & c_i \text{ covariant in } c \\ & | & [_{c_i}\ \overline{M}\ ]_{c_i} & c_i \text{ contravariant in } c \\ & | & M\ M & \\ & | & \mathsf{d} & \\ & | & \epsilon\,. & \end{array}$$

Since we have no directed edges, the production $M \leftarrow \mathsf{d}$ 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 $M \leftarrow [_{c_i}M]_{c_i}$. By induction, the M in the right-hand side represents a self-loop on a label ℓ. Thus the edges labelled by $[_{c_i}$ and $]_{c_i}$ arise from the same constructor edge rooted at ℓ and form a loop. The M-edge produced is a self-loop on the other end point of this loop, namely the ith child of ℓ. The case for $M \leftarrow [_{c_i}\overline{M}]_{c_i}$ 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:

$$\begin{tikzcd} \ell_1 \arrow[d, dashed, "\preceq^i_{\div}"] & \ell_2 \arrow[d, dashed, "\preceq^i_{+}"] \\ \ell'_1 \arrow[r, dotted, "M"] & \ell'_2 \end{tikzcd}$$

where we have $\ell_1 \rightsquigarrow_{\mathsf{m}} \ell_2$. 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 ℓ'_1 forms a self-loop.

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

$$\begin{array}{rcl} S & \leftarrow & P\ N \\ P & \leftarrow & \mathsf{p}\ P \\ & | & \epsilon \\ N & \leftarrow & \mathsf{n}\ N \\ & | & \epsilon\,. \end{array}$$

This grammar only accepts sequences formed by the regular expression $\mathsf{p}^*\mathsf{n}^*$. 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* The restrictions on POLYEQ mean that 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 no longer need to 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. In the paper Fähndrich *et al.* (2000), we showed, as a practical application, how to use POLYEQ to compute points-to information and function pointer information for the C programming language, and that the technique scales to large programs.

4. Discussion and related work

4.1. *Polymorphic subtyping*

Our work owes much to Mossin's work on flow analysis based on structural polymorphic subtyping systems (Mossin 1996). In particular, we build on the soundness proof provided there for $\mathsf{POLYFLOW}_{\mathsf{copy}}$, and use the same idea of introducing formal joins of labels to prove soundness or $\mathsf{POLYFLOW}_{\mathsf{CFL}}$ with respect to $\mathsf{POLYFLOW}_{\mathsf{copy}}$.

After the publication of Rehof and Fähndrich (2001), we learned of the concurrent work by Gustavsson and Svenningsson (Gustavsson and Svenningsson 2001), which addresses the copying problem of polymorphic constraint systems using a ‘constraint abstraction’ that serves a similar role to instantiation constraints. They were able to formulate a cubic algorithm without using CFL-reachability. Their work, however, does not seem to address the issue of computing flow to labels in nested contexts, such as to a particular formal parameter, nor does it extend to computing flow over type-polymorphic systems such as $\mathsf{POLYTYPE}_{\mathsf{CFL}}$.

4.2. *Higher-order context-sensitivity*

Type-polymorphic analyses are context sensitive in the type abstraction graph of a program, and not in the actual function call graph as is standard in first-order precise inter-procedural analyses. 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.

4.3. *Precise interprocedural dataflow analysis*

The work of Reps, Horwitz and Sagiv (Reps *et al.* 1995; Melski and Reps 2000) provided the connection between CFL-reachability problems and interprocedural, context-sensitive analysis. Reps *et al.* (1995) deals with *context-sensitive* (interprocedurally precise) analysis of first-order programs manipulating atomic data. Well-matched paths are used to select interprocedurally valid call–return sequences. Melski and Reps (2000) contains a higher-order but context-*in*sensitive analysis. Well-matched paths are used for *data-dependence analysis*, that is, to model cancellation properties of data constructors and destructors to track the flow of data through data-structures. Reps (2000) showed that the combination of the two techniques – context-sensitive data-dependence analysis – results in an uncomputable analysis problem. In contrast, our type-based techniques are concerned with 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 through finite approximation using recursive types that are unfolded only to finite depth. The techniques of Horwitz *et al.* (1995) and Reps (1998) for answering flow queries on demand transfer to our setting through our CFL-reachability formulation.

In Reps *et al.* (1995), the framework is flow-sensitive, that is, 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, that is, there is no concept of a store that is updated. There are two avenues open for enabling $\mathsf{POLYFLOW}_{\mathsf{CFL}}$ to deal with imperative languages. Either $\mathsf{POLYFLOW}_{\mathsf{CFL}}$ handles imperative features in the same way as ML's type system handles references, namely, by treating them in a flow-insensitive way, that is, all updates to a particular abstract location are accumulated. Alternatively, the program could be transformed into an interprocedural SSA form. $\mathsf{POLYFLOW}_{\mathsf{CFL}}$ 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, which is 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, $\mathsf{POLYFLOW}_{\mathsf{CFL}}$ derives a single global set of $|D|^2$ data-flow facts, namely the reachability relation between labels. Alternatively, we can view each label ℓ occurring in the program expression e as a program point. Then the set of data-flow facts derived for point ℓ is the set of reachable labels L_i for each label ℓ_i appearing in the type labelled by ℓ. However, the analogy does not express the fact that the paths in the exploded supergraph of Reps *et al.* (1995) 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.

```
letrec P = λa.λb.
          if0 a then
             let b₁ = c^ℓc in
             let a₁ = (a - b₁)^ℓa₁ in
             let q = P^j a₁ b₁ in
             let b₂ = q.2^ℓb₂ in
             q
           else p
in

let main = λx. P^i x 0^ℓ0
```

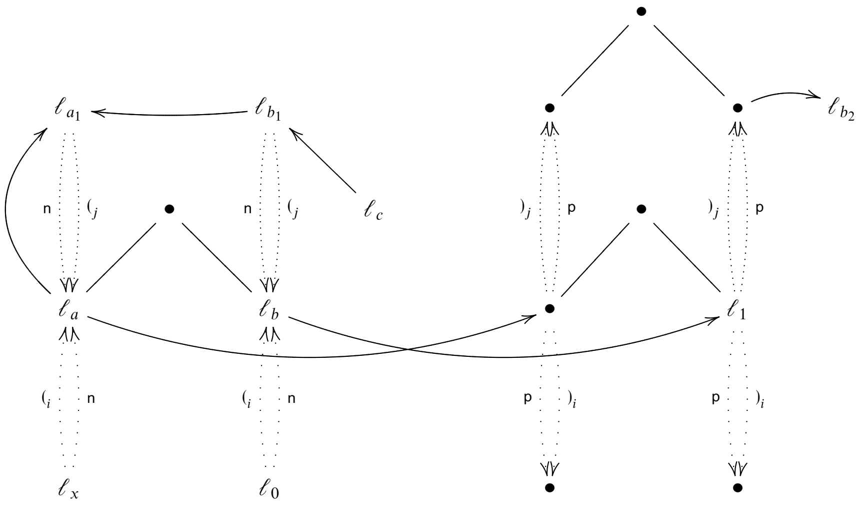


Fig. 29. Example from Reps *et.al,* (1995) (some edges omitted)

Figure 29 shows an SSA translation of the running example in Reps *et al.* (1995) 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 ℓ_0 to represent the uninitialised value, and in order to model how uninitialised 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 uninitialised after the recursive call to P at *j*. In our translation, the question is: Can ℓ_0 flow to ℓ_{b_2}? 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 ℓ_0 with ℓ_{b_2} is

$$\ell_0 \xrightarrow{(_i} \ell_b \rightarrow \ell_1 \xrightarrow{)_j} \bullet \rightarrow \ell_{b_2},$$

but it is invalid, since $(_i$ does not match $)_j$. The other three paths are not accepted by the grammar either. Thus, there is no flow from ℓ_0 to ℓ_{b_2} and we can conclude that b_2 is initialised on all paths.

4.4. *Summarisation in context-sensitive analyses*

Many context-sensitive flow analyses based on function summaries (see, for example, Chatterjee *et al.* (1999), Liang and Harrold (1999) and Foster *et al.* (2000)) are presented as two-phase computations. In phase 1, information is propagated from the callees (where it originates) to the callers. In $\mathsf{POLYFLOW}_{\mathsf{CFL}}$, 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, we do not need to compute two global phases. Instead, we can use demand-driven algorithms. In the global 2-phase approach, demand-driven algorithms can be used easily 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 two-phase algorithm (Mossin 1996, pages 90–91). His formulation relies crucially on the fact that the types include a set of polymorphic flow constraints that are *copied* on instantiation, including constraints of the form $L \leqslant \ell$, 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 reanalysing 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 (see, for example, Jagannathan and Wright (1995) and Nielson and Nielson (1997)).

4.5. *Constraint simplification*

Program analyses based on polymorphic subtyping depend crucially on constraint simplification in order to be at all practical (Fuh and Mishra 1988; Fähndrich and Aiken 1996a; Pottier 1996; Mossin 1996; Flanagan and Felleisen 1997; Pottier 1998). In the particular case of Mossin's algorithm for inferring derivations of $\mathsf{POLYFLOW}_{\mathsf{copy}}$, the constraint set C

is simplified before forming the quantified type $\forall\vec{\ell}.C \Rightarrow \sigma$. 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 of retaining only those constraints in C that involve free labels or labels occurring in σ, while retaining all flow implied by C. This reduces to removing intermediate labels, while adding enough transitive constraints.

Since $\mathsf{POLYFLOW}_{\mathsf{CFL}}$ 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 σ and its instance; and
2 the transitive rule [Trans] together with rule [Match] in the flow relation $\vdash_{\mathsf{CFL}}$ derive the simplified constraints that are copied in $\mathsf{POLYFLOW}_{\mathsf{copy}}$ directly.

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 (Fuh and Mishra 1988) may be useful in obtaining smaller polymorphic types (that is, 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 (Fähndrich *et al.* 1998).

4.6. *Type-structure polymorphism*

Mossin touches briefly on the subject of flow analyses in the presence of polymorphic type structure (Mossin 1996). Expressed in our formulation, he considers only two distinct hole variables h^{α}_{+} and $h^{\alpha}_{\div}$ for each polymorphic type variable α, where the first is used to annotate positive occurrences of α, and the second for negative occurrences of α. Such an approach approximates the flow of values by assuming that all input occurrences of a type variable α flow to all output occurrences of α. For a function such as

$$\texttt{twice} : \forall\alpha.(\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$$

where `twice f a = f (f a)`, the treatment he proposes yields strictly less precise flow than our approach in POLYTYPE.

Reps (2000) provided 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 (O'Callahan and Jackson 1997). 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.

4.7. *Other related work*

Henglein's work on semi-unification and polymorphic recursion (Henglein 1993) provides the logical foundation for our work. Dussart *et al.* (1995a) presented an algorithm for binding time analysis using polymorphic recursion with subtyping constraints, which uses constraint copying.

The unpublished work Dussart *et al.* (1995b) formulates Dussart *et al.* (1995a) purely in terms of semi-unification. However, it does not not make the connection to CFL-reachability and does not address the computation of flow information. We believe that our formulation of $\mathsf{POLYFLOW}_{\mathsf{CFL}}$ 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 (Heintze and McAllester 1997). The system they study corresponds to a type-based flow analysis with subtyping but without polymorphism. Such a system has also been studied in Mossin (1996). 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_{CFL}, individual queries can be answered in linear time by ignoring the matching problem and treating all edges as unlabelled flow edges.

More recently, Pratikakis *et al.* (2005) investigated flow paths in the presence of existential types. In general, the combination of existential packages and polymorphism is undecidable, since existential packing and unpacking is analogous to data construction/destruction. To overcome this problem, flow paths involving existential introduction and elimination are not allowed to also have polymorphic instantiation edges – their approach replaces them with ordinary flow edges in that case. They found that this trade-off worked well in practice for proving correct locking in Java programs, where existentials are used to correlate locks with the data they protect.

4.8. *Open problems*

We believe that the present work gives rise to a number of interesting open problems:

— Proving soundness of the flow relation of system $\mathsf{POLYTYPE}$ with polymorphic recursion (see Section 3.1.2);
— Extending the system to incorporate non-structural subtyping, where the underlying type structure allows subtyping between types that do not have the same syntactic structure;
— Extending the system to incorporate recursive types (due to the undecidability result of Reps (2000), the problem must be restricted, for example, by considering only regular structures of recursive labelled types – a simple example of such a restriction would be to consider only finite unfoldings of recursive types);
— Giving similar characterisations of flow in more expressive polymorphic systems (an especially interesting case would be system F – second-order polymorphic lambda calculus).

5. Conclusions

We have 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 $O(n^8)$ 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 characterisation 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 (Reps *et al.* 1995) to the setting of type-based analysis, resulting in an algorithm that 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 inhibitors of scaling for such systems.

5.1. *Practical experience*

We have implemented numerous special cases of the general flow algorithm described in this article. The idea of using CFL techniques to compute context-sensitive flow information is indeed practical. In Section 3.2 we have already described POLYEQ (Fähndrich *et al.* 2000), which is a special case where flow edges are undirected. Performance numbers for POLYEQ appear in the corresponding conference paper. The most general implementation we have attempted so far is described in a technical report (Das *et al.* 2001), where CFL is combined with the one-level flow algorithm by Das (Das 2000). That implementation was able to analyse over two million lines of C in under three minutes. More detailed performance numbers appear in that technical report.

Appendix A. Soundness proof

In this appendix we prove the soundness of the system $\mathsf{POLYFLOW}_{\mathsf{CFL}}$. We will do so by proving the system sound relative to $\mathsf{POLYFLOW}_{\mathsf{copy}}$; since soundness has been established for a system sufficiently similar to $\mathsf{POLYFLOW}_{\mathsf{copy}}$ in Mossin (1996), the soundness of $\mathsf{POLYFLOW}_{\mathsf{CFL}}$ follows from the soundness results of Mossin (1996).

A.1. *Overview of the proof*

The flow relation for a program e determined by a typing $t;I;C;A \vdash_{\mathsf{CFL}} e : \sigma$ in $\mathsf{POLYFLOW}_{\mathsf{CFL}}$ is defined by the CFL-reachability relation $I;C \vdash \ell \rightsquigarrow \ell'$ 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 realised 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 (that is, 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 $\mathsf{POLYFLOW}_{\mathsf{copy}}$, which is sufficiently close to the system studied by Mossin, for which soundness has already been established (Mossin 1996). 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 of showing that, for every derivation in $\mathsf{POLYFLOW}_{\mathsf{CFL}}$, there exists a derivation in $\mathsf{POLYFLOW}_{\mathsf{copy}}$ such that the CFL-based flow relation defined by the derivation in $\mathsf{POLYFLOW}_{\mathsf{copy}}$ 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 $\mathsf{POLYFLOW}_{\mathsf{CFL}}$ derivation of $t;I;C;A \vdash_{\mathsf{CFL}} e : \sigma$. We wish to show that there exists a derivation $C_g{}';I';C';A' \vdash_{cp} e : \sigma'$ in $\mathsf{POLYFLOW}_{\mathsf{copy}}$ 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 for why the CFL-based system works is thus 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

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

where flow from ℓ_3 to ℓ_4 is recovered by *matched flow* under CFL-reachability. This intuition provides the proof idea that we can construct the desired typing derivation of $C_g{}';I';C';A' \vdash_{cp} e : \sigma'$ in $\mathsf{POLYFLOW}_{\mathsf{copy}}$ from the given derivation of $t;I;C;A \vdash_{\mathsf{CFL}} e : \sigma$ by *closing* the set C under matched flow, using the constraints in I and C. Let C^I denote this flow closure of C, that is, we take C^I to be the set of all flow constraints of the form $\ell \leqslant \ell'$ where $I;C \vdash_{\mathsf{CFL}} \ell \rightsquigarrow_{\mathsf{m}} \ell'$ holds. Then we will show that C^I contains all the constraint copies needed for a derivation in the copy-based system. The technical property needed here is that, whenever a substitution φ is used at an instantiation site, we have

$$C^I \vdash \varphi(C^I), \tag{3}$$

that is, C^I is closed under substitutions. This property will render C^I applicable in the rule [Inst] of $\mathsf{POLYFLOW}_{\mathsf{copy}}$.

$$\begin{array}{lrcl} \text{(Types)} & \tau & ::= & int \mid \tau \rightarrow \tau \mid \tau \times \tau \\ \text{(Labelled Types)} & \tau\langle s\rangle, \sigma & ::= & int^{\ell} \mid \sigma \rightarrow^{\ell} \sigma \mid \sigma \times^{\ell} \sigma \\ \text{(Labels)} & L & ::= & \ell \mid \bigsqcup_i \ell_i \end{array}$$

Fig. 30. Extended labelled types

$$\boxed{C \vdash^{\leqslant} \sigma \leqslant \sigma}$$

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

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

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

$$\frac{C \vdash \ell_i \leqslant \ell, i = 1 \ldots n}{C \vdash \bigsqcup_{i=1}^{n} \ell_i \leqslant \ell}\,[\text{LubL}]$$

$$\frac{i \in \{1, \ldots, n\}}{C \vdash \ell_i \leqslant \bigsqcup_{i=1}^{n} \ell_i}\,[\text{LubR}]$$

Fig. 31. Extended subtype relation

It turns out that in order for C^I to have this property, we require certain properties of the derivation of $t; I; C; A \vdash_{\mathsf{CFL}} e : \sigma$. Such derivations are called *normal derivations* in the proof below. However, it can be shown that normal derivations can be assumed without loss of generality since an arbitrary derivation in $\mathsf{POLYFLOW}_{\mathsf{CFL}}$ can be normalised.

The structure of the proof is as follows. We begin with a number of technical definitions in Section A.2. We then establish the property (3) in Section A.3 under certain assumptions about the elements of the judgement $t; I; C; A \vdash_{\mathsf{CFL}} e : \sigma$. The central notion here is that of a *normal instantiation context*, which is defined in Section A.2. We then show in Section A.4 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 Figures 30 and 31. 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 31. The extended language allows us to translate fixpoint types in polymorphic recursive typings of system $\mathsf{POLYFLOW}_{\mathsf{CFL}}$ into fixpoint types of system $\mathsf{POLYFLOW}_{\mathsf{copy}}$. A similar method was used in Mossin (1996) to guarantee the existence of type fixpoints in his system.

A.2. *Normal instantiation contexts*

Our goal in this section is to work towards the characterisation of *normal derivations*, which will be given in Definition 6, by establishing some auxiliary definitions and properties. The central notion here is that of a *normal instantiation context*, Definition 5. The overall idea is that a normal derivation is one in which all applications of rule [Inst] have a canonical form with respect to the data (called the instantiation context) that are relevant for determining the instantiation performed. Naturally, those data include the constraint sets, type assumptions and type involved. A normal derivation, then, is one in which, informally speaking, all instantiation sites (uses of rule [Inst]) have normal instantiation contexts.

Definition 1 (Polarised constraint sets). Let C be a set of flow constraints, σ be a labelled type and $F \subseteq fl(\sigma)$. We say that C *is polarised with respect to* σ *and* F, written $C \rhd_F \sigma$, if and only if the following conditions hold for all $\ell \in F$:

1 Whenever $C \vdash \ell \leqslant \ell'$ with $\ell \neq \ell'$, we have ℓ occurs negatively in σ.
2 Whenever $C \vdash \ell' \leqslant \ell$ with $\ell \neq \ell'$, we have ℓ occurs positively in σ.

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

Definition 3 (CFL closure). Let C be a set of flow constraints and 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

$$C^I = \{\ell \leqslant \ell' \mid I; C \vdash \ell \rightsquigarrow_{\mathsf{m}} \ell'\}.$$

That is, C^I is the relation of matched CFL flow, considered as a flow constraint set. Note that, by rule [Level] of Figure 11, we have $C \subseteq C^I$ for all I.

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

$$\mathbf{L}(\ell, C, I, A, \sigma) = \{\ell' \in fl(A) \cup fl(\sigma) \mid C^I \vdash \ell' \leqslant \ell\}.$$

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

$$\varphi_{\langle C,I,A,\sigma\rangle}(\ell) = \begin{cases} \varphi(\ell) & \text{if } \ell \in gen(A, \sigma) \\ \bigsqcup \varphi\mathbf{L}(\ell, C, I, A, \sigma) & \text{if } \ell \in gen(A, \sigma, C) \setminus gen(A, \sigma). \end{cases}$$

We can now define the notion of a normal instantiation context.

Definition 5 (Normal instantiation context). We say that an instantiation context $\langle C, I, A, \sigma, \varphi\rangle$ is a *normal instantiation context at index i*, if and only if the following conditions hold:

(i) $dom\, \varphi = gen(A, \sigma)$.

(ii) $C \rhd_{dom\,\varphi} \sigma$.
(iii) $I \vdash \sigma \preceq^i_+ \sigma' : \varphi$.
(iv) $\forall \ell \in fl(A).\ I \vdash \ell \preceq^i_+ \ell$ and $I \vdash \ell \preceq^i_{\div} \ell$.
(v) Whenever $I \vdash \ell_1 \preceq^i_p \ell_2$ and $I \vdash \ell_3 \preceq^j_{p'} \ell_4$ with $\ell_1 \neq \ell_2$ and $\ell_3 \neq \ell_4$, we have $\ell_2 \neq \ell_3$.
(vi) $fl(C) = fl(C^I)$.

A few comments on the meaning and use of the conditions apearing in Definition 5 are in order here. Condition (i) simply says that the domain of φ contains all and only necessary data. Condition (ii) expresses the idea that constraint set C only contains relations that are forced by the polarities of occurrences of labels in the target type σ, and hence C does not contain any unnecessary constraints. Condition (iii) states that the substitution φ is associated with the instantiation relation at the top level at instantiation site i (and hence, the relation has positive polarity). Condition (iv) expresses the idea that instantiation acts as the identity on labels appearing in the assumption set. Condition (v) captures non-interference between labels, which can be attained by a suitable hygienic renaming of labels in a derivation. Finally, condition (vi) asserts that the closure of C does not contain any unnecessary data.

These conditions can be satisfied by suitably transforming a typing derivation, and capture the essence of a normal form derivation. In the proof of Lemma A.5, we will transform typing proofs in such a way that all of these conditions are satisfied.

In outline, the rest of the proof is as follows. Lemma A.1 is a step used in the proof of Lemma A.2. Lemma A.2, in turn, is used to establish Theorem A.3, which is used in the proof of Lemma A.4. Finally, in the proof of Theorem A.6, Lemma A.4 and Lemma A.5 are composed to yield the desired soundness result.

Lemma A.1. If $\langle C, I, A, \sigma, \varphi \rangle$ is a normal instantiation context at index i, then

$$C^I \rhd_{dom\,\varphi} \sigma\ .$$

Proof. Consider the first condition for $C^I \rhd_{dom\,\varphi} \sigma$ from Definition 1. Suppose that $\ell \in dom\,\varphi$ and $C^I \vdash \ell \leqslant \ell'$. We then must show that ℓ occurs negatively in σ. For this, it is sufficient to prove that ℓ occurs negatively in σ whenever $I ; C \vdash_{\mathsf{CFL}} \ell \rightsquigarrow_{\mathsf{m}} \ell'$ with $\ell \in dom\,\varphi$. We prove this property by induction on the proof of $I ; C \vdash_{\mathsf{CFL}} \ell \rightsquigarrow_{\mathsf{m}} \ell'$, taking cases over the last rule used in the proof:

— For the base case, suppose that the last rule used is

$$\frac{C \vdash \ell \leqslant \ell'}{I ; C \vdash \ell \rightsquigarrow_{\mathsf{m}} \ell'}\,[\text{Level}]$$

Because $\langle C, I, A, \sigma, \varphi \rangle$ is a normal instantiation context, we have $C \rhd_{dom\,\varphi} \sigma$, so $C \vdash \ell \leqslant \ell'$ implies that ℓ occurs negatively in σ.

— Suppose that the last rule used is

$$\frac{I ; C \vdash \ell \rightsquigarrow_{\mathsf{m}} \ell_1 \qquad I ; C \vdash \ell_1 \rightsquigarrow_{\mathsf{m}} \ell'}{I ; C \vdash \ell \rightsquigarrow_{\mathsf{m}} \ell'}\,[\text{Trans}]$$

By the induction hypothesis, $I ; C \vdash \ell \rightsquigarrow_{\mathsf{m}} \ell_1$ implies that ℓ occurs negatively in σ.

— Suppose that the last rule used is

$$\frac{I \vdash \ell_1 \preceq^j_{\div} \ell \qquad I;C \vdash \ell_1 \rightsquigarrow_{\mathsf{m}} \ell_2 \qquad I \vdash \ell_2 \preceq^j_{+} \ell'}{I;C \vdash \ell \rightsquigarrow_{\mathsf{m}} \ell'}\,[\text{Match}]$$

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

There are no other rules applicable to derive $I;C \vdash_{\mathsf{CFL}} \ell \rightsquigarrow_{\mathsf{m}} \ell'$, so the proof of the desired property is complete.

Proving the second property of Definition 1 is analogous and omitted here. □

Lemma A.2. Suppose that $\langle C, I, A, \sigma, \varphi\rangle$ is a normal instantiation context at index i, and suppose, furthermore, that ℓ_0 and ℓ_1 are labels in $fl(C)$ such that one of the following conditions is satisfied for each $j = 0, 1$:

(c1) $\ell_j \in gen(A, \sigma)$; or
(c2) $\ell_j \notin gen(A, \sigma, C)$.

Then $\varphi(\ell_0) \leqslant \varphi(\ell_1) \in \varphi(C^I)$ implies $I;C \vdash_{\mathsf{CFL}} \varphi(\ell_0) \rightsquigarrow_{\mathsf{m}} \varphi(\ell_1)$.

Proof. Note that since $\ell_0, \ell_1 \in fl(C)$, we have that $\ell_j \notin gen(A, \sigma, C)$ implies $\ell_j \in fl(A)$. Also, since $\varphi(\ell_0) \leqslant \varphi(\ell_1) \in \varphi(C^I)$, we have $\ell_0 \leqslant \ell_1 \in C^I$. Finally, because $\langle C, I, A, \sigma, \varphi\rangle$ is normal, Lemma A.1 yields that $C^I \rhd_{dom\,\varphi} \sigma$.

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): $\ell_0 \notin gen(A, \sigma, C), \ell_1 \notin gen(A, \sigma, C)$.
Then $\ell_0 = \ell, \ell_1 = \ell'$, and we have $\ell \leqslant \ell' \in C$, from which we get $I;C \vdash \ell \rightsquigarrow_{\mathsf{m}} \ell'$ by rule [Level].

— *Case* (c1) – (c2): $\ell_0 \in gen(A, \sigma), \ell_1 \notin gen(A, \sigma, C)$.
In this case, we have $\ell_1 = \varphi(\ell_1) = \ell'$. Since $\ell_0 \in gen(A, \sigma)$ with $\ell_0 \leqslant \ell_1 \in C^I$, it follows from $C^I \rhd_{dom\,\varphi} \sigma$ that ℓ_0 occurs negatively in σ. It then follows from the assumption that $\langle C, I, A, \sigma, \varphi\rangle$ is normal (condition (iii) of Definition 5) that we have

$$I \vdash \ell_0 \preceq^i_{\div} \ell\,. \tag{4}$$

Since $\ell_1 \in fl(C)$ and $\ell_1 \notin gen(A, \sigma, C)$, 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 $I \vdash \ell_1 \preceq^i_p \ell_1$ for $p = +$ and $p = \div$. Choosing this relation with $p = +$, we have $I \vdash \ell_1 \preceq^i_+ \ell_1$. Because $\ell_1 = \ell'$, this yields

$$I \vdash \ell' \preceq^i_+ \ell'\,. \tag{5}$$

Since $\ell_0 \leqslant \ell_1 \in C^I$, it follows that we have $I\,; C \vdash \ell_0 \rightsquigarrow_{\mathsf{m}} \ell_1$. Because $\ell' = \ell_1$, we then have

$$I\,; C \vdash \ell_0 \rightsquigarrow_{\mathsf{m}} \ell'\,. \tag{6}$$

Composing (4), (5) and (6), we get, by rule [Match], $I\,; C \vdash \ell \rightsquigarrow_{\mathsf{m}} \ell'$, as desired.

— *Case* $(c2) - (c1)$: $\ell_0 \notin gen(A, \sigma, C), \ell_1 \in gen(A, \sigma)$.
In this case we have $\ell_0 = \varphi(\ell_0) = \ell$. Since $\ell_1 \in gen(A, \sigma)$ with $\ell_0 \leqslant \ell_1 \in C^I$, it follows from $C^I \rhd_{dom\,\varphi} \sigma$ that ℓ_1 occurs positively in σ. It then follows from the assumption that $\langle C, I, A, \sigma, \varphi\rangle$ is normal (condition (iii) of Definition 5) that we have

$$I \vdash \ell_1 \preceq^i_+ \ell'\,. \tag{7}$$

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 $I \vdash \ell_0 \preceq^i_p \ell_0$ for $p = +$ and $p = \div$. Choosing this relation with $p = \div$, we have $I \vdash \ell_0 \preceq^i_\div \ell_0$. Because $\ell_0 = \ell$, this yields

$$I \vdash \ell \preceq^i_\div \ell\,. \tag{8}$$

Since $\ell_0 \leqslant \ell_1 \in C^I$, it follows that we have $I\,; C \vdash \ell_0 \rightsquigarrow_{\mathsf{m}} \ell_1$. Because $\ell = \ell_0$, we then have

$$I\,; C \vdash \ell \rightsquigarrow_{\mathsf{m}} \ell_1\,. \tag{9}$$

Composing (7), (8) and (9), we get, by rule [Match], $I\,; C \vdash \ell \rightsquigarrow_{\mathsf{m}} \ell'$, as desired.

— *Case* $(c1) - (c1)$: $\ell_0 \in gen(A, \sigma), \ell_1 \in gen(A, \sigma)$.
Since $\ell_0 \in dom\,\varphi$ and $\ell_1 \in dom\,\varphi$ with $\ell_0 \leqslant \ell_1 \in C^I$, it follows from $C^I \rhd_{dom\,\varphi} \sigma$ that ℓ_0 occurs negatively in σ and ℓ_1 occurs positively in σ. It then follows from the assumption that $\langle C, I, A, \sigma, \varphi\rangle$ is normal (condition (iii) of Definition 5) that we have

$$I \vdash \ell_0 \preceq^i_\div \ell \tag{10}$$

and

$$I \vdash \ell_1 \preceq^i_+ \ell'\,. \tag{11}$$

Since we also have

$$I\,; C \vdash \ell_0 \rightsquigarrow_{\mathsf{m}} \ell_1 \tag{12}$$

by $\ell_0 \leqslant \ell_1 \in C^I$, we can apply rule [Match] to (10), (11) and (12) to get $I\,; C \vdash_{\mathsf{CFL}} \ell \rightsquigarrow_{\mathsf{m}} \ell'$, as desired. □

Corollary A.1. Suppose that $\langle C, I, A, \sigma, \varphi\rangle$ is a normal instantiation context at index i, and let ℓ_0 and ℓ_1 be as stated in Lemma A.2. Then $\varphi(C^I) \vdash \varphi(\ell_0) \leqslant \varphi(\ell_1)$ implies $I\,; C \vdash_{\mathsf{CFL}} \varphi(\ell_0) \rightsquigarrow_{\mathsf{m}} \varphi(\ell_1)$.

Proof. The proof is by induction on the proof of $\varphi(C^I) \vdash \varphi(\ell_0) \leqslant \varphi(\ell_1)$, using Lemma A.2 for the base case and reflexivity and transitivity of the relation $\rightsquigarrow_{\mathsf{m}}$ (using rules [Level] and [Trans] of Figure 11) for the inductive cases. □

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 we have

$$C^I \vdash \widehat{\varphi}(C^I)\,.$$

Proof. We will show that whenever $L \leqslant L' \in \widehat{\varphi}(C^I)$, we have $C^I \vdash L \leqslant L'$. So we assume for arbitrary L, L' that $L \leqslant L' \in \widehat{\varphi}(C^I)$. Then for some $\ell_0, \ell_1 \in fl(C^I) = fl(C)$, we have $\ell_0 \leqslant \ell_1 \in C^I$ 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 ℓ_j, $j = 0, 1$, there are three disjoint cases to consider:

(*c*1) $\ell_j \in gen(A, \sigma)$
(*c*2) $\ell_j \in gen(A, \sigma, C) \setminus gen(A, \sigma)$
(*c*3) $\ell_j \notin gen(A, \sigma, C)$.

This yields 9 possible cases for ℓ_0 and ℓ_1. We will prove $C^I \vdash L \leqslant L'$ by considering each of these 9 possible cases. Note that $gen(A, \sigma) = dom(\varphi)$ and $gen(A, \sigma, C) = dom(\widehat{\varphi})$, and thus $gen(A, \sigma, C) \setminus gen(A, \sigma) = dom(\widehat{\varphi}) \setminus dom(\varphi)$.

First consider all cases where ℓ_0 and ℓ_1 are either in $gen(A, \sigma) = dom(\varphi)$ or not in $gen(A, \sigma, C) = dom(\widehat{\varphi})$. These are just the combinations of the cases (*c*1) and (*c*3) above, and in these cases we have $\widehat{\varphi} = \varphi$, and Lemma A.2 gives the conclusion

$$I\,; C \vdash L \rightsquigarrow_{\mathsf{m}} L'$$

in all of these cases. It then follows, by definition of C^I, that $C^I \vdash L \leqslant L'$ in all these cases. We therefore only need to consider the remaining 5 combinations of (*c*1), (*c*2), (*c*3) that are not combinations of (*c*1) and (*c*3):

— *Case* (*c*1) − (*c*2): $\ell_0 \in gen(A, \sigma), \ell_1 \in gen(A, \sigma, C) \setminus gen(A, \sigma)$.
In this case, by Definition 4, we have

$$L' = \widehat{\varphi}(\ell_1) = \bigsqcup \varphi\mathbf{L}(\ell_1, C, I, A, \sigma) \tag{13}$$

and

$$L = \widehat{\varphi}(\ell_0) = \varphi(\ell_0)\,. \tag{14}$$

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

$$\varphi(\ell_0) \in \varphi\mathbf{L}(\ell_1, C, I, A, \sigma)\,. \tag{15}$$

By (15) together with rule [LubR], we have $\vdash \varphi(\ell_0) \leqslant \bigsqcup \mathbf{L}(\ell_1, C, I, A, \sigma)$. Using (13) and (14), we can conclude $\vdash \widehat{\varphi}(\ell_0) \leqslant \widehat{\varphi}(\ell_1)$, and hence $C^I \vdash L \leqslant L'$, as desired.

— *Case* (*c*2) − (*c*1): $\ell_0 \in gen(A, \sigma, C) \setminus gen(A, \sigma)$.
In this case we have

$$L = \widehat{\varphi}(\ell_0) = \bigsqcup \varphi\mathbf{L}(\ell_0, C, I, A, \sigma) \tag{16}$$

and

$$L' = \widehat{\varphi}(\ell_1) = \varphi(\ell_1)\,. \tag{17}$$

Since $\ell_1 \in gen(A, \sigma)$, we have $\ell_1 \in fl(\sigma)$. Because $\ell_0 \leqslant \ell_1 \in C^I$, we have

$$\forall \ell'' \in \mathbf{L}(\ell_0, C, I, A, \sigma).\ C^I \vdash \ell'' \leqslant \ell_1\,. \tag{18}$$

Now let ℓ'' be an arbitrary label in $\mathbf{L}(\ell_0, C, I, A, \sigma)$. Since $\vdash$ is preserved under substitutions, we can conclude $\varphi(C^I) \vdash \varphi(\ell'') \leqslant \varphi(\ell_1)$ from (18). By definition of $\mathbf{L}(\ell_0, C, I, A, \sigma)$, we have $\ell'' \in fl(A) \cup fl(\sigma)$, and hence either $\ell'' \in gen(A, \sigma)$ or $\ell'' \notin gen(A, \sigma, C)$. Moreover, $\ell_1 \in gen(A, \sigma)$. Corollary A.1 is therefore applicable, and allows us to conclude

$$\forall \ell'' \in \mathbf{L}(\ell_0, C, I, A, \sigma).\ I\,; C \vdash_{\mathsf{CFL}} \varphi(\ell'') \rightsquigarrow_{\mathsf{m}} \varphi(\ell_1). \tag{19}$$

From (19), we can conclude that $\varphi(\ell'') \leqslant \varphi(\ell_1) \in C^I$ for all $\ell'' \in \mathbf{L}(\ell_0, C, I, A, \sigma)$. By rule [LubL], we then get $C^I \vdash \bigsqcup \varphi\mathbf{L}(\ell_0, C, I, A, \sigma) \leqslant \varphi(\ell_1)$, hence $C^I \vdash L \leqslant L'$ follows from (16) and (17), which proves the theorem in this case.

— *Case* $(c2) - (c2)$: $\ell_0 \in gen(A, \sigma, C) \setminus gen(A, \sigma), \ell_1 \in gen(A, \sigma, C) \setminus gen(A, \sigma)$.
In this case we have

$$L = \widehat{\varphi}(\ell_0) = \bigsqcup \varphi\mathbf{L}(\ell_0, C, I, A, \sigma) \tag{20}$$

and

$$L' = \widehat{\varphi}(\ell_1) = \bigsqcup \varphi\mathbf{L}(\ell_1, C, I, A, \sigma). \tag{21}$$

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

$$\varphi\mathbf{L}(\ell_0, C, A, \sigma) \subseteq \varphi\mathbf{L}(\ell_1, C, A, \sigma). \tag{22}$$

From (22), together with an application of rule [LubR] followed by an application of rule [LubL], we get

$$\vdash \bigsqcup \varphi\mathbf{L}(\ell_0, C, I, A, \sigma) \leqslant \bigsqcup \varphi\mathbf{L}(\ell_1, C, I, A, \sigma). \tag{23}$$

But (23), (20) and (21) together show that $\vdash L \leqslant L'$, hence $C^I \vdash L \leqslant L'$, as desired.

— *Case* $(c3) - (c2)$: $\ell_0 \notin gen(A, \sigma, C), \ell_1 \in gen(A, \sigma, C) \setminus gen(A, \sigma)$.
In this case we have

$$L = \widehat{\varphi}(\ell_0) = \varphi(\ell_0) = \ell_0 \tag{24}$$

and

$$L' = \widehat{\varphi}(\ell_1) = \bigsqcup \varphi\mathbf{L}(\ell_1, C, I, A, \sigma). \tag{25}$$

Note that because $\ell_0 \notin gen(A, \sigma, C)$, we must have $\ell_0 \in fl(A)$. Since $\ell_0 \leqslant \ell_1 \in C^I$, it follows that $\ell_0 \in \mathbf{L}(\ell_1, C, I, A, \sigma)$, hence $\varphi(\ell_0) \in \varphi\mathbf{L}(\ell_1, C, I, A, \sigma)$, and thus $\vdash \varphi(\ell_0) \leqslant \bigsqcup \varphi\mathbf{L}(\ell_1, C, I, A, \sigma)$ by rule [LubR]. Using (24) and (25), we can conclude $\vdash L \leqslant L'$, and thus $C^I \vdash L \leqslant L'$, as desired.

— *Case* $(c2) - (c3)$: $\ell_0 \in gen(A, \sigma, C) \setminus gen(A, \sigma), \ell_1 \notin gen(A, \sigma, C)$.
In this case we have

$$L = \widehat{\varphi}(\ell_0) = \bigsqcup \varphi\mathbf{L}(\ell_0, C, I, A, \sigma) \tag{26}$$

and

$$L' = \widehat{\varphi}(\ell_1) = \varphi(\ell_1) = \ell_1. \tag{27}$$

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

$$\forall \ell'' \in \mathbf{L}(\ell_0, C, I, A, \sigma).\ \varphi(C^I) \vdash \varphi(\ell'') \leqslant \varphi(\ell_1)\,. \tag{28}$$

Let ℓ'' be an arbitrary label in $\mathbf{L}(\ell_0, C, I, A, \sigma)$. 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 A.1 is applicable to ℓ'' and ℓ_1 because $\ell_1 \notin gen(A,\sigma,C)$. From (28), we then conclude from Corollary A.1 that

$$\forall \ell'' \in \mathbf{L}(\ell_0, C, I, A, \sigma).\ I\,; C \vdash_{\mathsf{CFL}} \varphi(\ell'') \leadsto_{\mathsf{m}} \varphi(\ell_1)\,. \tag{29}$$

In turn, it follows from (29) that $\varphi(\ell'') \leqslant \varphi(\ell_1) \in C^I$ for all $\ell'' \in \mathbf{L}(\ell_0, C, I, A, \sigma)$. By rule [LubL], we can then conclude that $C^I \vdash \bigsqcup \varphi\mathbf{L}(\ell_0, C, I, A, \sigma) \leqslant \varphi(\ell_1)$, thereby showing $C^I \vdash \widehat{\varphi}(\ell_0) \leqslant \widehat{\varphi}(\ell_1)$, and hence (using (27) and (28)) we get $C^I \vdash L \leqslant L'$, as desired. □

A.4. *Normal derivations*

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

$$\frac{I \vdash \sigma \preceq^i_+ \sigma' : \varphi \qquad dom\,\varphi = \vec{\ell} \qquad I \vdash s \preceq^i_+ s \qquad I \vdash s \preceq^i_{\div} s}{t; I\,; C\,; A, f : (\forall \vec{\ell}.\sigma, s) \vdash_{\mathsf{CFL}} f^i : \sigma'}\,[\text{Inst}]$$

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

$$t; I\,; C\,; A, f : (\forall \vec{\ell}.\sigma_1, t) \vdash_{\mathsf{CFL}} e_2 : \sigma_2$$

is referred to as the *binding premise for* f within $\mathscr{D}$. We use $env(\mathscr{D}, f)$ to denote the environment A in this rule application (notice that we have $\vec{\ell} = gen(env(\mathscr{D}, f), \sigma_1)$ in the binding premise for f).

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

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

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

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

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

Let $\mathscr{D}$ be a derivation of the judgement

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

We translate a set of type assumptions A into $A^{\mathscr{D}}$ given by

$$\begin{aligned}\varnothing^{\mathscr{D}} &= \varnothing\\ (A, x : \sigma)^{\mathscr{D}} &= A^{\mathscr{D}}, x : \sigma\\ (A, f : (\forall\vec{\ell}.\sigma, t))^{\mathscr{D}} &= A^{\mathscr{D}}, f : (\forall\vec{\ell}.\sigma)^{\langle I,C,env(\mathscr{D},f)\rangle}\,.\end{aligned}$$

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

$$I^{\mathscr{D}} = I \cup \{\ell \preceq^i_+ \varphi_{\langle C,I,A,\sigma\rangle}(\ell) \mid \langle C,A,\sigma,\varphi\rangle = Instcon(\mathscr{D},i)\}\,.$$

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

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

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

is derivable by a normal derivation $\mathscr{D}$ in system $\mathsf{POLYFLOW_{CFL}}$. Then the judgement

$$C^I;I^{\mathscr{D}};C^I;A^{\mathscr{D}} \vdash_{cp} e : \sigma$$

is derivable in system $\mathsf{POLYFLOW_{copy}}$.

Proof. Let $\mathscr{D}$ be a normal derivation of the judgement $t;I;C;A \vdash_{\mathsf{CFL}} e : \sigma$ in system $\mathsf{POLYFLOW_{CFL}}$. We translate every $\mathsf{POLYFLOW_{CFL}}$-judgement of the form

$$t_0;I_0;C_0;A_0 \vdash_{\mathsf{CFL}} e_0 : \sigma_0$$

in $\mathscr{D}$ into the $\mathsf{POLYFLOW_{copy}}$-judgement

$$C_0^{I_0};I_0^{\mathscr{D}};C_0^{I_0};A_0^{\mathscr{D}} \vdash_{cp} e_0 : \sigma_0\,.$$

The derivation $\mathscr{D}$ thereby translates into a series of judgements, $\mathscr{D}'$, in $\mathsf{POLYFLOW_{copy}}$. We can then verify, by induction on the derivation $\mathscr{D}$, that the translated sequence of judgements $\mathscr{D}'$ constitutes a valid derivation in $\mathsf{POLYFLOW_{copy}}$. The proof is by cases over the last rule applied in $\mathscr{D}$. We will only do a few illustrative cases:

— Suppose that the last rule used in $\mathscr{D}$ is [Inst] of the form

$$\frac{I \vdash \sigma \preceq^i_+ \sigma' : \varphi \qquad dom\,\varphi = \vec{\ell} \qquad I \vdash s \preceq^i_+ s \qquad I \vdash s \preceq^i_{\div} s}{t;I;C;A, f : (\forall\vec{\ell}.\sigma, s) \vdash_{\mathsf{CFL}} f^i : \sigma'}\,[\text{Inst}]$$

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

$$I^{\mathscr{D}} \vdash \sigma \preceq^{i}_{+} \sigma' : \widehat{\varphi}. \tag{30}$$

Also, since the context $\langle I, C, A', \varphi, \sigma\rangle$ is normal by assumption, Theorem A.3 shows that we have

$$C^I \vdash \widehat{\varphi}(C^I). \tag{31}$$

Finally, by the definition of $A^{\mathscr{D}}$ (Definition 7), we have

$$(A, f : (\forall \ell.\sigma, s))^{\mathscr{D}} = A^{\mathscr{D}}, f : \forall \vec{\ell'}.C^I \Rightarrow \sigma \tag{32}$$

with $\vec{\ell'} = gen(env(A,f), \sigma, C)$. It then follows from (30), (31) and (32) that the following application of rule [Inst] is valid in $\mathsf{POLYFLOW}_{\mathsf{copy}}$:

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

This proves the lemma in this case.

— Suppose that the last rule used in $\mathscr{D}$ is [Sub] of the form

$$\frac{t; I; C; A \vdash_{\mathsf{CFL}} e : \sigma \qquad C \vdash \sigma \leqslant \sigma'}{t; I; C; A \vdash_{\mathsf{CFL}} e : \sigma'}\,[\text{Sub}]$$

By induction, we have $C^I; I^{\mathscr{D}}; C^I; A^{\mathscr{D}} \vdash_{cp} e : \sigma$ derivable in $\mathsf{POLYFLOW}_{\mathsf{copy}}$. Since $C \subseteq C^I$, we have $C^I \vdash \sigma \leqslant \sigma'$ by $C \vdash \sigma \leqslant \sigma'$. Hence, rule [Sub] is applicable in $\mathsf{POLYFLOW}_{\mathsf{copy}}$, and the conclusion yields $C^I; I^{\mathscr{D}}; C^I; A^{\mathscr{D}} \vdash_{cp} e : \sigma'$, which proves the lemma in this case.

— Suppose that the last rule used in $\mathscr{D}$ is an application of the rule [Let] of the form

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

By induction, we know that

$$C^I; I^{\mathscr{D}}; C^I; A^{\mathscr{D}} \vdash_{cp} e_1 : \sigma_1$$

is derivable, and

$$C^I; I^{\mathscr{D}}; C^I; A^{\mathscr{D}}, f : \forall \vec{\ell'}.C^I \Rightarrow \sigma_1 \vdash_{cp} e_2 : \sigma_2$$

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

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

This proves the lemma in this case.

— Suppose that the last rule used in $\mathscr{D}$ is an application of the rule [Rec] of the form

$$\frac{\begin{array}{ll} t;I;C;A,f : (\forall\vec{\ell}.\sigma_1,t) \vdash_{\mathsf{CFL}} e_1 : \sigma_1 & \vec{\ell} = gen(A,\sigma_1) \\ t;I;C;A,f : (\forall\vec{\ell}.\sigma_1,t) \vdash_{\mathsf{CFL}} e_2 : \sigma_2 & \vdash \tau\langle s\rangle : \sigma_1 \end{array}}{t;I;C;A \vdash_{\mathsf{CFL}} \texttt{letrec } f{:}\tau \texttt{ = } e_1 \texttt{ in } e_2 : \sigma_2}\,[\text{Rec}]$$

By induction, we know that

$$C^I;I^{\mathscr{D}};C^I;A^{\mathscr{D}},f : \forall\vec{\ell'}.C^I \Rightarrow \sigma_1 \vdash_{cp} e_1 : \sigma_1$$

is derivable in $\mathsf{POLYFLOW}_{\mathsf{copy}}$, and

$$C^I;I^{\mathscr{D}};C^I;A^{\mathscr{D}},f : \forall\vec{\ell'}.C^I \Rightarrow \sigma_1 \vdash_{cp} e_2 : \sigma_2$$

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

$$\frac{\begin{array}{l} C^I;I^{\mathscr{D}};C^I;A^{\mathscr{D}},f : \forall\vec{\ell'}.C^I \Rightarrow \sigma_1 \vdash_{cp} e_1 : \sigma_1 \\ C^I;I^{\mathscr{D}};C^I;A^{\mathscr{D}},f : \forall\vec{\ell'}.C^I \Rightarrow \sigma_1 \vdash_{cp} e_2 : \sigma_2 \\ C^I \subseteq C^I \qquad \vec{\ell'} = gen(A,\sigma_1,C^I) \qquad \vdash \tau\langle s\rangle : \sigma_1 \end{array}}{C^I;I^{\mathscr{D}};C^I;A^{\mathscr{D}} \vdash_{cp} \texttt{letrec } f{:}\tau \texttt{ = } e_1 \texttt{ in } e_2 : \sigma_2}\,[\text{Rec}]$$

This proves the lemma in this case.

The remaining cases are obvious and omitted here. □

A.5. *Existence of normal derivations*

Definition 9 (Label closure). For labelled types σ_1,σ_2, we define the *label closure* of the inequality $\sigma_1 \leqslant \sigma_2$, denoted $lcl(\sigma_1 \leqslant \sigma_2)$, to be the set of flow constraints implied by the constraint $\sigma_1 \leqslant \sigma_2$, as follows:

— $lcl(int^{\ell_1} \leqslant int^{\ell_2}) = \{\ell_1 \leqslant \ell_2\}$
— $lcl(\sigma_1 \times^{\ell_1} \sigma_2 \leqslant \sigma_3 \times^{\ell_2} \sigma_4) = \{\ell_1 \leqslant \ell_2\} \cup lcl(\sigma_1 \leqslant \sigma_3) \cup lcl(\sigma_2 \leqslant \sigma_4)$
— $lcl(\sigma_1 \to^{\ell_1} \sigma_2 \leqslant \sigma_3 \to^{\ell_2} \sigma_4) = \{\ell_1 \leqslant \ell_2\} \cup lcl(\sigma_3 \leqslant \sigma_1) \cup lcl(\sigma_2 \leqslant \sigma_4)$.

Note that we have

$$lcl(\sigma_1 \leqslant \sigma_2) \vdash \sigma_1 \leqslant \sigma_2 .$$

Lemma A.5 (Existence of normal derivations). For every derivation $\mathscr{D}$ of a judgement

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

in $\mathsf{POLYFLOW}_{\mathsf{CFL}}$ there exist C' and I' and a normal derivation $\mathscr{D}_N$ of

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

such that for all ℓ,ℓ' in $fl(e)$ we have

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

Proof. The proof is in 3 steps:

(I) We first introduce transformations on derivations, which transforms a given derivation $\mathscr{D}$ into a new derivation $\mathscr{D}_N$.

(II) We then show that the resulting derivation $\mathscr{D}_N$ is indeed normal.

(III) Finally, we show that the property

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

holds.

We can assume without loss of generality that the given derivation $\mathscr{D}$ satisfies the folowing *binding properties*:

— Any two program variables bound by distinct `let` or `letrec` bindings are distinct.

— Whenever $f : (\forall\vec{\ell}.\sigma, t)$ and $g : (\forall\vec{\ell'}.\sigma', t')$ are two distinct type assumptions occurring in distinct *binding premises* (Definition 6) of $\mathscr{D}$, the labels in $\vec{\ell}$ and $\vec{\ell'}$ are disjoint.

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

We now proceed with the proof:

(I) Let a derivation $\mathscr{D}$ of the judgement $t;I;C;A \vdash_{\mathsf{CFL}} e : \sigma$ be given in $\mathsf{POLYFLOW}_{\mathsf{CFL}}$. To obtain $\mathscr{D}_N$ from $\mathscr{D}$, we perform local transformations on every application of rule [Let] and [Rec] in $\mathscr{D}$. To describe the transformations of derivations, we need a few technical preliminaries.

For each `let` or `letrec` bound variable f in $\mathscr{D}$, let $\mathit{TyBind}(\mathscr{D}, f) = \sigma$, where σ is the uniquely determined type such that the binding $f : (\forall\vec{\ell}.\sigma, t)$ occurs in a binding premise within $\mathscr{D}$, and let $\mathit{LabBind}(\mathscr{D}, f)$ denote the set of labels in $\vec{\ell}$, where $\vec{\ell}$ is the vector of labels bound in that binding premise. Note that the *binding properties* assumed for $\mathscr{D}$ ensure that we have

$$\mathit{LabBind}(\mathscr{D}, f) \cap \mathit{LabBind}(\mathscr{D}, g) = \varnothing$$

for any f and g occurring in distinct binding premises in $\mathscr{D}$. Fix a `let` or `letrec` bound variable f in $\mathscr{D}$. Let ψ_f be a renaming on $\mathit{LabBind}(\mathscr{D}, f)$ mapping the labels in $\mathit{LabBind}(\mathscr{D}, f)$ to distinct labels that are to be kept *fresh* throughout $\mathscr{D}_N$ such that for any substitutions φ_1, φ_2 applied at any instantiation sites in $\mathscr{D}_N$, we have

$$\mathit{dom}\,\varphi_1 \cap \mathit{ran}\ \varphi_2 = \varnothing \tag{33}$$

where we set $\mathit{ran}\ \varphi = \{\varphi(\ell) \mid \ell \in \mathit{dom}\,\varphi\}$. Using our assumption that $\mathscr{D}$ satisfies the *binding properties* mentioned above, it is clear that ψ_f can be chosen in such a way that these conditions are satisfied. The property (33) can be obtained by a suitable renaming of bound variables and applications of the subtyping rule.

For each $\sigma \in \mathit{TyBind}(\mathscr{D}, f)$, let $\sigma^f = \psi_f(\sigma)$ and define the constraint set C^f by

$$C^f = C \cup \mathit{lcl}(\sigma \leqslant \sigma^f),$$

and define the set of instantiations constraints I^f by

$$I^f = \{\psi_f(\ell) \preceq^i_p \ell' \mid \ell \preceq^i_p \ell' \in I\}.$$

— **Transformation of applications of rule [Let]**:
Consider an untransformed application of rule [Let] within $\mathscr{D}$ of the form

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

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

$$\frac{\dfrac{t;I^f;C^f;A \vdash_{\mathsf{CFL}} e_1 : \sigma_1 \qquad C^f \vdash \sigma_1 \leqslant \sigma_1^f}{t;I^f;C^f;A \vdash_{\mathsf{CFL}} e_1 : \sigma_1^f}\,[\text{Sub}] \qquad \mathbf{J}}{t;I^f;C^f;A \vdash_{\mathsf{CFL}} \texttt{let } f \texttt{ = } e_1 \texttt{ in } e_2 : \sigma_2}\,[\text{Let}]$$

where

$$\mathbf{J} = t;I^f;C^f;A,f:(\forall\vec{\ell'}.\sigma_1^f,t) \vdash_{\mathsf{CFL}} e_2 : \sigma_2$$

and $\vec{\ell'} = gen(A,\sigma_1^f)$. This series of judgements enters into a valid subderivation of $\mathsf{POLYFLOW}_{\mathsf{CFL}}$. To see this, first suppose we have

$$\frac{t;I^f;C^f;A \vdash_{\mathsf{CFL}} e_1 : \sigma_1 \qquad C^f \vdash \sigma_1 \leqslant \sigma_1^f}{t;I^f;C^f;A \vdash_{\mathsf{CFL}} e_1 : \sigma_1^f}\,[\text{Sub}]$$

because $lcl(\sigma_1 \leqslant \sigma_1^f) \subseteq C^f$ by definition of C^f. Then, since, by assumption, we can derive

$$t;I;C;A,f:(\forall\vec{\ell}.\sigma_1,t) \vdash_{\mathsf{CFL}} e_2 : \sigma_2,$$

we can also derive

$$t;I^f;C^f;A,f:(\forall\vec{\ell'}.\sigma_1^f,t) \vdash_{\mathsf{CFL}} e_2 : \sigma_2$$

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 then follows that we can apply the [Let] rule as follows:

$$\frac{\begin{array}{l} t;I^f;C^f;A \vdash_{\mathsf{CFL}} e_1 : \sigma_1^f \qquad \vec{\ell'} = gen(A,\sigma_1^f) \\ t;I^f;C^f;A,f:(\forall\vec{\ell'}.\sigma_1^f,t) \vdash_{\mathsf{CFL}} e_2 : \sigma_2 \end{array}}{t;I^f;C^f;A \vdash_{\mathsf{CFL}} \texttt{let } f \texttt{ = } e_1 \texttt{ in } e_2 : \sigma_2}\,[\text{Let}]$$

— **Transformation of applications of rule [Rec]**:
Consider an untransformed application of rule [Rec] within $\mathscr{D}$ of the form

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

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

$$\dfrac{\dfrac{t;I^f;C^f;A,f:(\forall\vec{\ell'}.\sigma_1^f,t)\vdash_{\mathsf{CFL}} e_1:\sigma_1 \qquad C^f\vdash\sigma_1\leqslant\sigma_1^f}{t;I^f;C^f;A,f:(\forall\vec{\ell'}.\sigma_1^f,t)\vdash_{\mathsf{CFL}} e_1:\sigma_1^f}\,[\mathrm{Sub}] \qquad \mathbf{J}}{t;I^f;C^f;A\vdash_{\mathsf{CFL}} \mathtt{letrec}\ f\ \mathtt{=}\ e_1\ \mathtt{in}\ e_2:\sigma_2}\,[\mathrm{Rec}]$$

where

$$\mathbf{J} = t;I^f;C^f;A,f:(\forall\vec{\ell'}.\sigma_1^f,t)\vdash_{\mathsf{CFL}} e_2:\sigma_2$$

and $\vec{\ell'} = gen(A,\sigma_1^f)$. This series of judgements enters into a valid subderivation of $\mathsf{POLYFLOW_{CFL}}$. To see this, first suppose we have

$$t;I^f;C^f;A,f:(\forall\vec{\ell'}.\sigma_1^f,t)\vdash_{\mathsf{CFL}} e_1:\sigma_1$$

derivable, because (by assumption) we can derive

$$t;I;C;A,f:(\forall\vec{\ell}.\sigma_1,t)\vdash_{\mathsf{CFL}} e_1:\sigma_1\,.$$

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

$$\dfrac{t;I^f;C^f;A,f:(\forall\vec{\ell'}.\sigma_1^f,t)\vdash_{\mathsf{CFL}} e_1:\sigma_1 \qquad C^f\vdash\sigma_1\leqslant\sigma_1^f}{t;I^f;C^f;A,f:(\forall\vec{\ell'}.\sigma_1^f,t)\vdash_{\mathsf{CFL}} e_1:\sigma_1^f}\,[\mathrm{Sub}]$$

because $lcl(\sigma_1\leqslant\sigma_1^f)\subseteq C^f$ by definition of C^f. Finally, since by assumption we can derive

$$t;I;C;A,f:(\forall\vec{\ell}.\sigma_1,t)\vdash_{\mathsf{CFL}} e_2:\sigma_2\,,$$

we can also derive

$$t;I^f;C^f;A,f:(\forall\vec{\ell'}.\sigma_1^f,t)\vdash_{\mathsf{CFL}} e_2:\sigma_2$$

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

$$\dfrac{\begin{array}{l} t;I^f;C^f;A,f:(\forall\vec{\ell'}.\sigma_1^f,t)\vdash_{\mathsf{CFL}} e_0:\sigma_1^f \qquad \vec{\ell'}=gen(A,\sigma_1^f)\\ t;I^f;C^f;A,f:(\forall\vec{\ell'}.\sigma_1^f,t)\vdash_{\mathsf{CFL}} e_2:\sigma_2\end{array}}{t;I^f;C^f;A\vdash_{\mathsf{CFL}} \mathtt{letrec}\ f\ \mathtt{=}\ e_1\ \mathtt{in}\ e_2:\sigma_2}\,[\mathrm{Rec}]$$

By repeated application of the two proof transformations we can construct a transformed derivation $\mathscr{D}'$ of

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

Finally, in order to turn $\mathscr{D}'$ into a normal derivation $\mathscr{D}_N$, we add the trivial inequality $\ell\leqslant\ell$ to C^* of $\mathscr{D}'$ for every label ℓ occuring in I^* but not in C^*. Let C' be the resulting constraint set:

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

Finally, set $I'=I^*$.

(II) We now argue that the resulting derivation $\mathscr{D}_N$ of the judgement

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

is indeed normal. To do this, let $instcon(\mathscr{D}_N, i) = \langle C', I', A_i, \sigma, \varphi \rangle$ be an arbitrary instantiation context in $Instcon(\mathscr{D}_N)$. We have $A_i = env(\mathscr{D}_N, f)$ and $\sigma = \sigma_0^{(\mathscr{D},f)}$ for some σ_0 and f at instantiation site f^i. We must show the following properties (Definition 5):

(i) $dom\,\varphi = gen(A_i, \sigma)$.

(ii) $C' \rhd_{dom\,\varphi}\, \sigma$.

(iii) $I' \vdash \sigma \preceq^i_+ \sigma' : \varphi$.

(iv) $\forall \ell \in fl(A_i).\ I' \vdash \ell \preceq^i_+ \ell$ and $I' \vdash \ell \preceq^i_\div \ell$.

(v) Whenever $I' \vdash \ell_1 \preceq^i_p \ell_2$ and $I' \vdash \ell_3 \preceq^j_{p'} \ell_4$ with $\ell_1 \neq \ell_2$ and $\ell_3 \neq \ell_4$, we have $\ell_2 \neq \ell_3$.

(vi) $fl(C') = fl((C')^{I'})$.

with $\sigma = \sigma^f$.

We consider each property in turn:

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

(ii) $C' \rhd_{dom\,\varphi}\, \sigma$ 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 $A = A_i, f : (\forall \vec{\ell'}.\sigma^f, t)$, where $dom\,\varphi = gen(A_i, \sigma^f) = \vec{\ell'}$. By construction of $\mathscr{D}_N$, it is the case that C' contains exactly the set of flow constraints $lcl(\sigma \leqslant \sigma^f)$, where the flow variables in $\vec{\ell'}$ are all distinct and occur only in σ^f. It follows directly from this property that $C' \rhd_{dom\,\varphi}\, \sigma^f$.

(iii) $I' \vdash \sigma \preceq^i_+ \sigma' : \varphi$ follows from the rules of $\mathsf{POLYFLOW_{CFL}}$, for suitable σ'.

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

(v) Whenever $I' \vdash \ell_1 \preceq^i_p \ell_2$ and $I' \vdash \ell_3 \preceq^j_{p'} \ell_4$ with $\ell_1 \neq \ell_2$ and $\ell_3 \neq \ell_4$, we have $\ell_2 \neq \ell_3$. This property follows from (33) as follows. If $I' \vdash \ell_1 \preceq^i_p \ell_2$, then ℓ_1 is in the domain of φ, provided $\ell_1 \neq \ell_2$. Therefore, ℓ_2 is in the range of φ. Similarly, if $\ell_3 \neq \ell_4$, then ℓ_3 is in the domain of some substitution φ' used at instantiation site j. Hence, if $\ell_2 = \ell_3$, then the label ℓ_2 is in the domain of φ'. But ℓ_2 is also in the range of φ. This contradicts property (33) of the construction of $\mathscr{D}_N$ and is therefore impossible.

(vi) $fl(C') = fl((C')^{I'})$ 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 $fl(I') \subseteq fl(C')$, the property follows.

(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 $\mathscr{D}_N$: the set C' was constructed from C by addition of inequalities of the form $\sigma \leqslant \sigma^f$, where σ^f arises from σ 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

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

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

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

derivable in $\mathsf{POLYFLOW}_{\mathsf{copy}}$ such that for all labels ℓ and ℓ' ocurring in e we have

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

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

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

derivable by a normal derivation $\mathscr{D}$ in $\mathsf{POLYFLOW}_{\mathsf{CFL}}$ and such that for $\ell, \ell' \in fl(e)$ we have

$$I'; C' \vdash \ell \rightsquigarrow \ell' \Rightarrow I; C \vdash \ell \rightsquigarrow \ell' . \tag{34}$$

By Lemma A.4, the judgement

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

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

$$(I')^{\mathscr{D}}; (C')^{I'} \vdash_{cp} \ell \rightsquigarrow \ell' \Rightarrow I'; C' \vdash_{\mathsf{CFL}} \ell \rightsquigarrow \ell' . \tag{35}$$

Using (34) and (35) together, we get

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

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

References

Aiken, A., Wimmers, E. L. and Palsberg, J. (1997) Optimal representations of polymorphic types with subtyping. In: Proceedings of the International Symposium on Theoretical Aspects of Computer Science. *Springer-Verlag Lecture Notes in Computer Science* **1281** 47–76.

Chatterjee, R., Ryder, B. G. and Landi, W. A. (1999) Relevant context inference. In: *Conference Record of the 26th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.*

Curtis, P. (1990) Constrained quantification in polymorphic type analysis. Technical Report CSL-90-1, Xerox Palo Alto Research Center.

Das, M. (2000) Unification-based pointer analysis with directional assignments. In: *PLDI '00: Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, ACM Press 35–46.

Das, M., Fähndrich, M., Liblit, B. and Rehof, J. (2001) Estimating the impact of scalable pointer analysis on optimization. Technical Report MSR-TR-2001-20, Microsoft Research.

Dussart, D., Henglein, F. and Mossin, C. (1995a) Polymorphic recursion and subtype qualifications: Polymorphic binding-time analysis in polynomial time. In: Mycroft, A. (ed.) Proc. 2nd Int'l Static Analysis Symposium (SAS), Glasgow, Scotland. *Springer-Verlag Lecture Notes in Computer Science* **983** 118–135.

Dussart, D., Henglein, F. and Mossin, C. (1995b) Polymorphic recursion and subtype qualifications: Polymorphic binding-time analysis in polynomial time. (Unpublished draft.)

Eifrig, J., Smith, S. and Trifonov, V. (1995) Sound polymorphic type inference for objects. In: *Proceedings OOPSLA '95*.

Fähndrich, M. and Aiken, A. (1996a) Making set-constraint based program analyses scale. In: *First Workshop on Set Constraints at CP'96*, Cambridge, MA. (Available as Technical Report CSD-TR-96-917, University of California at Berkeley.)

Fähndrich, M. and Aiken, A. (1996b) Making set-constraint program analyses scale. In: *Workshop on Set Constraints*, Cambridge MA.

Fähndrich, M., Foster, J. S., Su, Z. and Aiken, A. (1998) Partial online cycle elimination in inclusion constraint graphs. In: Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation. *SIGPLAN notices* **33** (5) 85–96.

Fähndrich, M., Rehof, J. and Das, M. (2000) Scalable context-sensitive flow analysis using instantiation constraints. In: *Programming Language Design and Implementation.*

Fähndrich, M., Rehof, J. and Das, M. (2000) Scalable context-sensitive flow analysis using instantiation constraints. In: Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation. *SIGPLAN notices* 253–263.

Flanagan, C. and Felleisen, M. (1997) Componential set-based analysis. In: Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation. *SIGPLAN notices* **32** (6) 235–248.

Foster, J. S., Fähndrich, M. and Aiken, A. (2000) Polymorphic versus Monomorphic Flow-insensitive Points-to Analysis for C. In: Proceedings of the 7th International Static Analysis Symposium. *Springer-Verlag Lecture Notes in Computer Science* **1824**.

Fuh, Y.-C. and Mishra, P. (1988) Type inference with subtypes. In: *Proceedings of the 1988 European Symposium on Programming* 94–114.

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

Gustavsson, J. and Svenningsson, J. (2001) Constraint abstractions. In: Program as Data Objects: International Conference on the Theory and Application of Cryptographic Techniques. *Springer-Verlag Lecture Notes in Computer Science* **2053**.

Heintze, N. (1995) Control-flow analysis and type systems. In: Proceedings SAS '95, Second International Static Analysis Symposium, Glasgow, Scotland. *Springer-Verlag Lecture Notes in Computer Science* **983** 189–206.

Heintze, N. and McAllester, D. (1997) Linear-time subtransitive control flow analysis. In: Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation. *SIGPLAN notices* **32** (6) 261–272.

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

Horwitz, S., Reps, T. and Sagiv, M. (1995) Demand interprocedural dataflow analysis. In: ACM SIGSOFT Symposium on the Foundations of Software Engineering. *ACM SIGSOFT Software Engineering Notes* **20** (4) 104–115.

Jagannathan, S. and Wright, A. (1995) Effective flow analysis for avoiding run-time checks. In: Proceedings of the 2nd International Static Analysis Symposium. *Springer-Verlag Lecture Notes in Computer Science* **983** 207–224.

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

Liang, D. and Harrold, M. J. (1999) Efficient points-to analysis for whole-program analysis. In: *Proceedings of the 7th European Software Engineering Conference and the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering.*

Melski, D. and Reps, T. (1997) Interconvertibility of set constraints and context-free language reachability. In: Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM-97). *ACM SIGPLAN Notices* **32** (12) 74–89.

Melski, D. and Reps, T. (2000) Interconvertibility of set constraints and context-free language reachability. *Theoretical Computer Science* **248** (1-2) 29–98.

Mitchell, J. C. (1996) *Foundations for Programming Languages*, MIT Press.

Mossin, C. (1996) *Flow Analysis of Typed Higher-Order Programs*, Ph.D. thesis, DIKU, Department of Computer Science, University of Copenhagen.

Mycroft, A. (1984) Polymorphic type schemes and recursive definitions. In: *Proceedings of the 6th International Symposium on Programming* 217–228.

Nielson, F. and Nielson, H. R. (1997) Infinitary control flow analysis: a collecting semantics for closure analysis. In: *Conference Record of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM Press 332–345.

Nielson, F., Nielson, H. R. and Hankin, C. (1999) *Principles of Program Analysis*, Springer-Verlag.

O'Callahan, R. and Jackson, D. (1997) Lackwit: A program understanding tool based on type inference. In: *International Conference on Software Engineering.*

Palsberg, J. and O'Keefe, P. (1995) A type system equivalent to flow analysis. *Transactions on Programming Languages and Systems* **17** (4) 576–599.

Pessaux, F. (2000) *Détection Statique d'Exceptions non Rattrapée en Objective CAML*, Ph.D. thesis, Université Pierre et Marie Curie, Paris 6. (In French.)

Pottier, F. (1996) Simplifying subtyping constraints. In: Proceedings of the SIGPLAN '96 International Conference on Functional Programming (ICFP '96). *SIGPLAN notices* **31** (6) 122–133.

Pottier, F. (1998) *Type Inference in the Presence of Subtyping: From Theory to Practice*, Ph.D. thesis, Université Paris VII.

Pottier, F. and Rémy, D. (2005) The Essence of ML Type Inference. In: Pierce, B. (ed.) *Advanced Topics in Types and Programming*, MIT Press.

Pratikakis, P., Hicks, M. and Foster, J. S. (2005) Existential Label Flow Inference via CFL Reachability. Technical Report CS-TR-4700, University of Maryland, College Park. (Available at `http://hdl.handle.net/1903/3018`.)

Rehof, J. (1997) Minimal typings in atomic subtyping. In: *Symposium on Principles of Programming Languages.*

Rehof, J. (1998) *The Complexity of Simple Subtyping Systems*, Ph.D. thesis, Dept. of Computer Science, University of Copenhagen, Denmark.

Rehof, J. and Fähndrich, M. (2001) Type-based flow analysis: From polymorphic subtyping to CFL-reachability. In: *Proceedings POPL 2001, 28'th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.*

Reps, T. (1998) Program analysis via graph reachability. *Information and Software Technology* **40** (11-12) 701–726.

Reps, T. (2000) Undecidability of context-sensitive data-dependence analysis. *Transactions on Programming Languages and Systems* **22** (1) 162–186.

Reps, T., Horwitz, S. and Sagiv, M. (1995) Precise interprocedural dataflow analysis via graph reachability. In: *Conference record of POPL '95, 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* 49–61.

Rittri, M. (1995) Deriving dimensions under polymorphic recursion. In: *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, ACM Press 147–159.

Smith, G. S. (1994) Principal type schemes for functional programs with overloading and subtyping. *Science of Computer Programming* **23** (2-3) 197–226.

Trifonov, V. and Smith, S. (1996) Subtyping constrained types. In: Proceedings of the 3rd International Static Analysis Symposium. *Springer-Verlag Lecture Notes in Computer Science* **1145**.