Polyvariant Flow Analysis with Higher-ranked Polymorphic Types and Higher-order Effect Operators

Stefan Holdermans
Vector Fabrics
Paradijslaan 28, 5611 KN Eindhoven, The Netherlands
stefan@vectorfabrics.com

Jurriaan Hage
Dept. of Inf. and Comp. Sciences, Utrecht University
P.O. Box 80.089, 3508 TB Utrecht, The Netherlands
jur@cs.uu.nl

Abstract

We present a type and effect system for flow analysis that makes essential use of higher-ranked polymorphism. We show that, for higher-order functions, the expressiveness of higher-ranked types enables us to improve on the precision of conventional let-polymorphic analyses. Modularity and decidability of the analysis are guaranteed by making the analysis of each program parametric in the analyses of its inputs; in particular, we have that higher-order functions give rise to higher-order operations on effects. As flow typing is archetypical to a whole class of type and effect systems, our approach can be used to boost the precision of a wide range of type-based program analyses for higher-order languages.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Polymorphism; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program analysis; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Functional constructs, Type structure

General Terms Languages, Theory

Keywords type-based program analysis, higher-ranked polymorphism

1. Introduction

The use of polymorphic types in type and effect systems for static program analysis is usually limited to ML-style let-polymorphism. This restriction precludes the formal parameters of higher-order functions from being analysed polyvariantly rather than monovariantly. In this paper, we consider a type and effect system that allows analyses to be expressed in terms of higher-ranked polymorphic types and argue how the resulting polyvariant analyses are more powerful than the analyses obtained from let-polymorphic systems.

Specifically, our contributions are the following:

- We present an annotated type and effect system for flow analysis that makes essential use of higher-ranked polymorphism in both annotations and effects (Section 5). The resulting analysis is polyvariant in its treatment of lambda-bound variables, applicable to all well-typed terms in an explicitly typed lambda-calculus with Booleans and conditionals (Section 6.1), and sound with respect to an instrumented, flow-tracking semantics (Section 6.2).
- The main technical innovations of our system are its use of so-called *fully flexible types* to maintain the modularity of the analyses (Section 4.1) and its use of *annotation and effect operators* to have the analyses of higher-order functions explicitly parameterised in the analyses of their arguments (Section 4.2).
- For all terms with fully flexibly typed free variables, our system admits "best analyses" (Section 6.3), which can be obtained by means of a strikingly straightforward inference algorithm (Section 7).

We stress that flow typing is, in a sense, archetypical to a whole class of type and effect systems; as a wide range of other analyses, including binding-time analysis, strictness analysis, and usage analysis, are known to be expressible as variations of type-based control-flow analysis, we expect our approach to also apply to most if not all of these analyses.

2. Motivation

Numerous static program analyses depend on information about the flow of control in the program under analysis. Whereas for first-order languages this information is directly available from the program text, the situation for higher-order languages, in which functions or procedures can be passed as arguments to other functions or procedures, is considerably different; for these languages, one has to deal with the *dynamic dispatch problem*. Consider, for example, the following program fragment, written in some typed higher-order functional language:

$$h : (\texttt{bool} \to \texttt{bool}) \to \texttt{bool}$$
$$h\,f = \textbf{if}\, f\ \texttt{false}\ \textbf{then}\, f\ \texttt{true}\ \textbf{else}\ \texttt{false}.$$

As the function parameter f can, at run-time, be bound to any suitably typed function, it is not obvious to what code control is transferred when the condition f `false` in the body of h is evaluated.

To cope with the dynamic dispatch problem, several *flow analyses* have been proposed. Of particular interest are flow analyses that, in some way or another, take advantage of the structure that is imposed on programs by a static typing discipline for the language under analysis; such type-based analyses can typically be more effective than analyses for dynamically typed languages or analyses that ignore the well-typedness of analysed programs (Palsberg 2001). An important class of type-based analyses is then that of so-called *type and effect systems* that extend the typing disci-

ICFP'10, September 27–29, 2010, Baltimore, Maryland, USA.

plines of languages as to express properties beyond just plain data types (Nielson and Nielson 1999).

For instance, to track the flow of Boolean values through a program, we can decorate all occurrences of the Boolean constructors `false` and `true` in a program with labels $\ell_1, \ell_2, \ldots$, as in

$$h\,f = \textbf{if}\, f\, \texttt{false}^{\ell_1}\, \textbf{then}\, f\, \texttt{true}^{\ell_2}\, \textbf{else}\, \texttt{false}^{\ell_3},$$

and adopt an extended type system that annotates the type `bool` of Boolean values with sets of labels identifying the possible construction sites of these values. The Boolean identity function, $id\,x = x$, then, for example, can have the type $\texttt{bool}^{\{\ell_1,\ell_2\}} \to \texttt{bool}^{\{\ell_1,\ell_2\}}$, indicating that if its argument x is a Boolean constructed at any of the sites labelled with ℓ_1 or ℓ_2, then so is its result. Assigning the function *id* this type prepares it for being passed as an argument to the function h above, which can be of type $(\texttt{bool}^{\{\ell_1,\ell_2\}} \to \texttt{bool}^{\{\ell_1,\ell_2\}}) \to \texttt{bool}^{\{\ell_1,\ell_2,\ell_3\}}$. However, in general the assigned type is too specific as *id* could be used in other contexts as well. This is suggestive of annotating the argument and result types of *id* with a larger set as to reflect all uses of *id* in the program, but this is undesirable for at least two reasons. First, it requires the whole program to be available as information is required about all possible uses of *id* and thus precludes the analysis from being *modular*. Second, it renders the analysis of program fragments that directly or indirectly use *id* rather imprecise as the larger set shows up for every value that is obtained by applying *id*, irrespective of the actual argument supplied. This latter issue is known as the *poisoning problem* (Wansbrough and Peyton Jones 1999). In general, poisoning can be reduced by making the analysis more *polyvariant*, that is, allowing different uses of an identifier to be analysed independently.

One way to make an analysis based on a type and effect system both more modular and more polyvariant is by making use of *annotation polymorphism*. For example, *id* can be assigned the polymorphic type $\forall\beta.\texttt{bool}^{\beta} \to \texttt{bool}^{\beta}$ with β ranging over sets of constructor labels. Indeed, this type can be derived from just the definition of *id* and instantiated to a more specific type for each use of *id*.

The use of polymorphism in type and effect systems is usually limited to ML-style let-polymorphism (Damas and Milner 1982), meaning that polymorphic types can only be assigned to identifiers bound at top level or in local definitions. This seems like a natural restriction as program analyses are almost always required to be performed fully automatically and ML-style polymorphic types allow for mechanically and modularly deriving "best analyses", which are then typically defined in terms of *principal types*, whereas more expressive uses of polymorphism do not necessarily admit such mechanisation.

To see why we may still want to consider less restrictive uses of polymorphism, consider once more applying the function h from the example above to the Boolean identity function *id*. In a let-polymorphic type and effect system, h can be expected to have a type much like $\forall\beta.(\texttt{bool}^{\{\ell_1,\ell_2\}} \to \texttt{bool}^{\beta}) \to \texttt{bool}^{\beta\cup\{\ell_3\}}$. The aforementioned polymorphic type of *id* is then instantiated to $\texttt{bool}^{\{\ell_1,\ell_2\}} \to \texttt{bool}^{\{\ell_1,\ell_2\}}$ and instantiating the variable β in the type of h then yields $\texttt{bool}^{\{\ell_1,\ell_2,\ell_3\}}$ as the type obtained for the application $h\,id$. Note that this result is imprecise in the sense that the Boolean constructed at the site labelled with ℓ_1 never flows to the result of any invocation of h. This imprecision is caused by the restriction that, in an ML-style type and effect system, the formal parameter f of h has to be assigned a monomorphic type. Hence, uses of f in the body of h are analysed monovariantly and subjected to poisoning.

Now, if the type and effect system were to somehow allow the parameter f of h to have a polymorphic type, we could have

$$h : (\forall\beta.\texttt{bool}^{\beta} \to \texttt{bool}^{\beta}) \to \texttt{bool}^{\{\ell_2,\ell_3\}}$$

with different choices for β for different uses of f in the body of h allowing for a more polyvariant analysis. Here, we require h to have a so-called *rank-2 polymorphic type*. In general, the rank of a polymorphic type describes the maximum depth at which universal quantifiers occur in contravariant positions (Kfoury and Tiuryn 1992).

As it is well-known that the higher-ranked fragment of the polymorphic lambda-calculus does not admit principal types and that type inference is undecidable for rank 3 and higher, it is not immediately obvious that higher-ranked polymorphic types can be of any practical use in type and effect systems for fully automatic program analysis. However, here it is crucial that we only need to consider types that are polymorphic in the *annotations* that decorate types rather than in the types themselves. As it turns out, higher-ranked annotation polymorphism does indeed provide a feasible basis for attaining analyses that are fully polyvariant with respect to the formal parameters of higher-order functions.[1]

The main challenge of incorporating higher-ranked polymorphic types in a type and effect system is then to take advantage of their expressive power without compromising the modularity of the analysis. For example, the rank-2 type for h that was proposed above is too specific as it presumes that the function bound to the parameter f will manifest identity-like behaviour, which in general is obviously unacceptably restrictive. Below, we will rise to the challenge and present a modular type and effect system with higher-ranked polymorphic types that admits analyses for higher-order functions like h that are adaptive enough for all appropriately typed functions to be passed in as arguments, while still allowing for the formal parameters of these higher-order functions to be analysed polyvariantly.

3. Preliminaries

Throughout this paper, we use, as the language under analysis, an eagerly evaluated and simply typed Church-style lambda-calculus with Booleans, conditionals, and general recursion.

Assuming an abstract set of program labels and a countable infinite set of variable symbols,

$$\begin{array}{lcll} \ell & \in & \textbf{Lab} & \text{labels} \\ x & \in & \textbf{Var} & \text{variables,} \end{array}$$

terms in our language are constructed from variables, producers, and consumers; that is, we have

$$\begin{array}{lcll} t & \in & \textbf{Tm} & \text{terms} \\ p & \in & \textbf{Prod} & \text{producers} \\ c & \in & \textbf{Cons} & \text{consumers} \end{array}$$

with

$$\begin{array}{lcl} t & ::= & x \mid p^{\ell} \mid c^{\ell} \\ p & ::= & \texttt{false} \mid \texttt{true} \mid \lambda x : \tau.t_1 \\ c & ::= & \textbf{if}\, t_1\, \textbf{then}\, t_2\, \textbf{else}\, t_3 \mid t_1\, t_2 \mid \textbf{fix}\, t_1. \end{array}$$

All producers and consumers are labelled. A producer is either one of the Boolean constructors `false` and `true` or a lambda-abstraction, while consumers subsume conditionals, function applications, and fixed points. As usual, function application associates to the left and lambda-abstractions extend as far to the right as possible. Each abstraction is annotated with the type of its formal parameter, where types,

[1] This approach is reminiscent of the use of polymorphic recursion in the type-based binding-time analysis of Dussart et al. (1995): while polymorphic recursion in its full, untamed glory renders type inference undecidable, its restriction to binding-time annotations has proven to allow for a very expressive yet workable analysis. See Section 4.3.

Evaluation $\boxed{t \Downarrow_{\mathsf{F}} p^{\ell}}$

$$\frac{}{p^{\ell} \Downarrow_{\{\}} p^{\ell}}\ [\textit{e-prod}]$$

$$\frac{t_1 \Downarrow_{\mathsf{F}_1} \mathtt{true}^{\ell_{\mathrm{p}}} \quad t_2 \Downarrow_{\mathsf{F}_2} p^{\ell}}{(\mathbf{if}\ t_1\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3)^{\ell_{\mathrm{c}}} \Downarrow_{\mathsf{F}_1 \cup \{(\ell_{\mathrm{c}},\ell_{\mathrm{p}})\} \cup \mathsf{F}_2} p^{\ell}}\ [\textit{e-if-true}]$$

$$\frac{t_1 \Downarrow_{\mathsf{F}_1} \mathtt{false}^{\ell_{\mathrm{p}}} \quad t_3 \Downarrow_{\mathsf{F}_3} p^{\ell}}{(\mathbf{if}\ t_1\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3)^{\ell_{\mathrm{c}}} \Downarrow_{\mathsf{F}_1 \cup \{(\ell_{\mathrm{c}},\ell_{\mathrm{p}})\} \cup \mathsf{F}_3} p^{\ell}}\ [\textit{e-if-false}]$$

$$\frac{t_1 \Downarrow_{\mathsf{F}_1} (\lambda x : \tau.t_0)^{\ell_{\mathrm{p}}} \quad t_2 \Downarrow_{\mathsf{F}_2} p_2{}^{\ell_2} \quad [x \mapsto p_2{}^{\ell_2}]t_0 \Downarrow_{\mathsf{F}_0} p^{\ell}}{(t_1\ t_2)^{\ell_{\mathrm{c}}} \Downarrow_{\mathsf{F}_1 \cup \mathsf{F}_2 \cup \{(\ell_{\mathrm{c}},\ell_{\mathrm{p}})\} \cup \mathsf{F}_0} p^{\ell}}\ [\textit{e-app}]$$

$$\frac{t_1 \Downarrow_{\mathsf{F}_1} (\lambda x : \tau.t_0)^{\ell_{\mathrm{p}}} \quad [x \mapsto (\mathbf{fix}\ t_1)^{\ell_{\mathrm{c}}}]t_0 \Downarrow_{\mathsf{F}_0} p^{\ell}}{(\mathbf{fix}\ t_1)^{\ell_{\mathrm{c}}} \Downarrow_{\mathsf{F}_1 \cup \{(\ell_{\mathrm{c}},\ell_{\mathrm{p}})\} \cup \mathsf{F}_0} p^{\ell}}\ [\textit{e-fix}]$$

Figure 1. Instrumented natural semantics.

Typing $\boxed{\Gamma \vdash t : \tau}$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}\ [\textit{t-var}]$$

$$\frac{}{\Gamma \vdash \mathtt{false}^{\ell} : \mathtt{bool}}\ [\textit{t-false}] \qquad \frac{}{\Gamma \vdash \mathtt{true}^{\ell} : \mathtt{bool}}\ [\textit{t-true}]$$

$$\frac{\Gamma[x \mapsto \tau_1] \vdash t_1 : \tau_2}{\Gamma \vdash (\lambda x : \tau_1.t_1)^{\ell} : \tau_1 \to \tau_2}\ [\textit{t-abs}]$$

$$\frac{\Gamma \vdash t_1 : \mathtt{bool} \quad \Gamma \vdash t_2 : \tau \quad \Gamma \vdash t_3 : \tau}{\Gamma \vdash (\mathbf{if}\ t_1\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3)^{\ell} : \tau}\ [\textit{t-if}]$$

$$\frac{\Gamma \vdash t_1 : \tau_2 \to \tau \quad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash (t_1\ t_2)^{\ell} : \tau}\ [\textit{t-app}] \qquad \frac{\Gamma \vdash t_1 : \tau \to \tau}{\Gamma \vdash (\mathbf{fix}\ t_1)^{\ell} : \tau}\ [\textit{t-fix}]$$

Figure 2. The underlying type system.

$$\tau \quad \in \quad \mathbf{Ty} \quad \text{types},$$

are given by

$$\tau \quad ::= \quad \mathtt{bool} \mid \tau_1 \to \tau_2.$$

An instrumented natural semantics is given in Figure 1 as a set of inference rules for deriving judgements of the form $t \Downarrow_{\mathsf{F}} p^{\ell}$, indicating that the term t evaluates in zero or more steps to the value produced by the ℓ-labelled producer p, while the flow of values during evaluation is captured by the *flow set* F,

$$\mathsf{F} \quad \in \quad \mathbf{Flow} \quad = \quad \mathscr{P}(\mathbf{Lab} \times \mathbf{Lab}) \quad \text{flow}.$$

Concretely, each pair $(\ell_{\mathrm{c}}, \ell_{\mathrm{p}})$ in a flow set F witnesses the consumption of a value produced at a program point labelled with ℓ_{p} by a consumer labelled with ℓ_{c}. Note that Boolean values (produced by the constructors `false` and `true`) are consumed by conditionals, while functions (produced by lambda-abstractions) are consumed by function applications and occurrences of the fixed-point operator. Evaluation proceeds under a call-by-value strategy; capture-avoiding substitution, in rules [*e-app*] en [*e-fix*], is denoted by $[\cdot \mapsto \cdot]\hat{}\_$.

The static semantics of the language is presented in Figure 2 in terms of typing rules for deriving judgements $\Gamma \vdash t : \tau$, expressing that, in the type environment Γ, the term t has the type τ. Here, type environments are finite maps from variables to types:

$$\Gamma \quad \in \quad \mathbf{TyEnv} \quad = \quad \mathbf{Var} \to_{\mathrm{fin}} \mathbf{Ty} \quad \text{type environments}.$$

In the sequel, we are only concerned with well-typed terms. The static semantics of Figure 2 is referred to as the *underlying type system* and the types from the underlying type system play a crucial rôle in our approach as they guide our polyvariant flow analysis.

4. Key Ideas

In this section, we discuss the key ideas behind the type and effect system that will be presented in Section 5. Recall that our main objective is to provide a modular flow analysis that allows lambda-bound variables to be analysed polyvariantly rather than monovariantly.

To this end, we associate with each term t in the program a triple $\widehat{\tau}^{\psi}\ \&\ \varphi$, consisting of an annotated type $\widehat{\tau}$, an annotation ψ, and an effect φ. The idea is that the annotation ψ describes the possible production sites of the values that t can evaluate to and that the effect φ describes the flow that may be incurred from the evaluation of t. Thus, annotations are essentially sets of labels ℓ, while effects are sets of pairs (ℓ, ψ) consisting of a consumer label ℓ and an annotation ψ. Annotated types are constructed from the type `bool` of Booleans and annotated function types of the form $\widehat{\tau}_1{}^{\psi_1} \xrightarrow{\varphi_0} \widehat{\tau}^{\psi_2}$, where ψ_1 and ψ_2 denote the production sites of, respectively, the argument and the result of a function, and φ_0 is the so-called *latent effect* of a function, i.e., the effect that may be observed from applying the function to an argument. Furthermore, and crucially, we allow universal quantification over both annotations and effects to occur anywhere in an annotated type.

4.1 Fully Flexible Types

As an example, consider the Boolean negation function produced by

$$(\lambda x : \mathtt{bool}.\,(\mathbf{if}\ x\ \mathbf{then}\ \mathtt{false}^{\ell_1}\ \mathbf{else}\ \mathtt{true}^{\ell_2})^{\ell_3})^{\ell_4}.$$

Analysing this function may then result in the triple

$$(\forall\beta.\mathtt{bool}^{\beta} \xrightarrow{\{(\ell_3,\beta)\}} \mathtt{bool}^{\{\ell_1,\ell_2\}})^{\{\ell_4\}}\ \&\ \{\},$$

expressing that the ℓ_4-labelled lambda-abstraction immediately (i.e., flowlessly) produces a function that may have its argument consumed by the conditional labelled with ℓ_3 before returning a Boolean that is produced at either ℓ_1 or ℓ_2. Note that the annotated type for the negation function is polymorphic in the annotation for its argument x and how this is crucial for obtaining an analysis that is modular: whatever Boolean it is applied to, the type of the function can always be instantiated to obtain a suitable analysis for the application.

As modularity is a key aspect of our analysis, let us from now on assume that functions are always analysed with maximum applicability in mind and, hence, that all functions have types that are indeed polymorphic in their argument annotations. We shall refer to such types as *fully flexible types*.

4.2 Annotation and Effect Operators

To demonstrate how the notion of fully flexible types extends to higher-order functions, let us consider the second-order function produced by

$$(\lambda f : \mathtt{bool} \to \mathtt{bool}.\,(f\ \mathtt{true}^{\ell_5})^{\ell_6})^{\ell_7},$$

which applies its argument to the Boolean `true` produced at ℓ_5.

How can we, for such a function, obtain an analysis that can be regarded as fully flexible? Clearly, modularity requires us to be polymorphic in the annotation of the argument function f. Moreover, as we assume that all functions have fully flexible types, the type of any function to be bound to f will itself be polymorphic in its argument annotation too, i.e., have a type of the form $\forall\beta.\mathtt{bool}^{\beta} \xrightarrow{\varphi} \mathtt{bool}^{\psi}$. In general, the latent effect φ and the result annotation ψ of f depend on the argument annotation β. We

can make this explicit by writing φ and ψ as functions of β: $\forall\beta.\,\texttt{bool}^{\beta} \xrightarrow{\varphi_0\,\beta} \texttt{bool}^{\psi_0\,\beta}$. If we allow annotation and effect abstraction in annotated types, then the annotated types for all functions of underlying type $\texttt{bool} \to \texttt{bool}$ can be written in this form. For instance, for the annotated type of the negation function from Section 4.1, we have $\varphi_0 = \lambda\beta'.\,\{(\ell_3,\beta')\}$ and $\psi_0 = \lambda\beta'.\,\{\ell_1,\ell_2\}$, yielding

$$\forall\beta.\,\texttt{bool}^{\beta} \xrightarrow{(\lambda\beta'.\,\{(\ell_3,\beta')\})\,\beta} \texttt{bool}^{((\lambda\beta'.\,\{\ell_1,\ell_2\})\,\beta)}.$$

Returning to the analysis of the second-order function as a whole, modularity once more requires us to assume a type for f that can be instantiated for all possible choices for φ_0 and ψ_0 and, hence, we end up with a triple consisting of the rank-2 type

$$\forall\beta_f.\forall\delta_0.\forall\beta_0.\\ (\forall\beta.\,\texttt{bool}^{\beta} \xrightarrow{\delta_0\,\beta} \texttt{bool}^{(\beta_0\,\beta)})^{\beta_f} \xrightarrow{\{(\ell_6,\beta_f)\}\cup\delta_0\,\{\ell_5\}} \texttt{bool}^{(\beta_0\,\{\ell_5\})},$$

the singleton annotation and $\{\ell_7\}$ and the empty effect $\{\}$. Here, the variables δ_0 and β_0 range over, respectively, effect and annotation *operators* rather than proper effects and annotations. Note how both the latent effect $\{(\ell_6,\beta_f)\}\cup\delta_0\,\{\ell_5\}$ and the result annotation $\beta_0\,\{\ell_5\}$ express that for any call of the second-order function, the polymorphic type of the function bound to its parameter f is instantiated with the annotation $\{\ell_5\}$ and that the supplied effect and annotation operators are applied accordingly.

Essentially, what we have done here amounts to parameterising the analysis of a function by the analyses of its arguments. For a first-order function, the analysis of an argument is captured by a single annotation that identifies its possible production sites. For a higher-order function, the analysis of an argument of function type is captured by a proper annotation that identifies the possible production sites of the supplied function, and effect and annotation operators that describe how the analysis of the argument function depends on the analyses for its own arguments.

Now, concretely, if we instantiate the annotated type of the second-order function above as to prepare it for being applied to the negation function from Section 4.1 and thus supply it with the analysis for the negation function, then, after beta-reducing the effects and annotations, we obtain the instantiated type

$$(\forall\beta.\,\texttt{bool}^{\beta} \xrightarrow{\{(\ell_3,\beta)\}} \texttt{bool}^{\{\ell_1,\ell_2\}})^{\{\ell_4\}} \\ \xrightarrow{\{(\ell_6,\{\ell_4\}),(\ell_3,\{\ell_5\})\}} \texttt{bool}^{\{\ell_1,\ell_2\}}.$$

As a final example of the use of annotation and effect operators, consider the higher-order abstraction (cf. the running example from Section 2)

$$(\lambda f : \texttt{bool} \to \texttt{bool}.\\ \quad (\textbf{if}\ (f\ \texttt{false}^{\ell_1})^{\ell_2}\ \textbf{then}\ (f\ \texttt{true}^{\ell_3})^{\ell_4}\ \textbf{else}\ \texttt{false}^{\ell_5})^{\ell_6})^{\ell_7}$$

and its fully flexible annotated type

$$\forall\beta_f.\forall\delta_0.\forall\beta_0.(\forall\beta.\,\texttt{bool}^{\beta} \xrightarrow{\delta_0\,\beta} \texttt{bool}^{(\beta_0\,\beta)})^{\beta_f} \\ \xrightarrow{\{(\ell_2,\beta_f)\}\cup\delta_0\,\{\ell_1\}\cup\{(\ell_6,\beta_0\,\{\ell_1\})\}\cup\{(\ell_4,\beta_f)\}\cup\delta_0\,\{\ell_3\}} \\ \texttt{bool}^{(\beta_0\,\{\ell_3\}\cup\{\ell_5\})},$$

and how this type can be instantiated with the analysis for the Boolean identity function produced by $(\lambda x : \texttt{bool}.x)^{\ell_8}$ to yield the desired polyvariant

$$(\forall\beta.\,\texttt{bool}^{\beta} \xrightarrow{\{\}} \texttt{bool}^{\beta}) \xrightarrow{\{(\ell_2,\ell_8),(\ell_6,\ell_1),(\ell_4,\ell_8)\}} \texttt{bool}^{\{\ell_3,\ell_5\}}.$$

4.3 Polymorphic Recursion

Being able to associate polymorphic annotated types with lambda-bound variables naturally induces polymorphic recursion (Mycroft 1984) for fixed points. Indeed, as recursive functions are constructed as fixed points $\textbf{fix}\,t_1$ of terms t_1 with higher-order types $(\tau_1 \to \tau_2) \to \tau_1 \to \tau_2$ and higher-ranked polymorphism allows for arguments to such t_1 to have polymorphic annotated types of the form $\forall\beta.\,\widehat{\tau}_1{}^{\beta} \xrightarrow{\varphi} \widehat{\tau}_2{}^{\psi}$, it follows that recursive calls, i.e., uses of its argument by t_1, may be analysed polyvariantly rather than monovariantly.

As expected, higher-ranked polymorphism gives you polymorphic recursion for free.

5. Flow Analysis with Higher-ranked Types

In this section, we present the details of our type and effect system for flow analysis with higher-ranked polymorphic types.

5.1 Annotations and Effects

We assume to have at our disposal countable infinite sets of annotation variables (ranged over by β) and effect variables (ranged over by δ):

$$\begin{array}{rcll} \beta & \in & \textbf{AnnVar} & \text{annotation variables} \\ \delta & \in & \textbf{EffVar} & \text{effect variables.} \end{array}$$

Annotations and effects are then given by

$$\begin{array}{rcll} \psi & \in & \textbf{Ann} & \text{annotations} \\ \varphi & \in & \textbf{Eff} & \text{effects} \end{array}$$

with

$$\begin{array}{rcl} \psi & ::= & \beta \mid \{\} \mid \{\ell\} \mid \lambda\beta :: s.\,\psi_1 \mid \psi_1\ \psi_2 \mid \psi_1 \cup \psi_2 \\ \varphi & ::= & \delta \mid \{\} \mid \{(\ell,\psi)\} \mid \lambda\beta :: s.\,\varphi_1 \mid \varphi_1\ \psi \\ & \mid & \lambda\delta :: s.\,\varphi_1 \mid \varphi_1\ \varphi_2 \mid \varphi_1 \cup \varphi_2. \end{array}$$

Note that annotations ψ may contain annotation abstractions $\lambda\beta :: s.\,\psi_1$ and annotation applications $\psi_1\ \psi_2$, while effects may contain annotation abstractions $\lambda\beta :: s.\,\varphi_1$ and annotation applications $\varphi_1\ \psi$ as well as effect abstractions $\lambda\delta :: s.\,\varphi_1$ and effect applications $\varphi_1\ \varphi_2$. Furthermore, note that abstractions over annotations and effects make mention of sorts s,

$$s \in \textbf{Sort} \quad \text{sorts.}$$

That is, to make sure that abstractions and applications in annotations and effects are used in meaningful ways only, we depend on sorts to act as the "types" of annotations and effects. Sorts are then constructed from

$$s ::= \texttt{ann} \mid \texttt{eff} \mid s_1 \to s_2,$$

where `ann` denotes the sort of proper annotations, `eff` the sort of proper effects, and $s_1 \to s_2$ the sort of operators that take annotations or effects of sort s_1 to annotations or effects of sort s_2. Storing the sorts of free annotation and effect variables in a sort environment Σ,

$$\Sigma \in \textbf{SortEnv} = (\textbf{AnnVar} \cup \textbf{EffVar}) \to_{\text{fin}} \textbf{Sort} \quad \text{sort env.},$$

which maps from annotation and effect variables to sorts, rules for assigning sorts to annotations and effects can be given as in Figure 3.

In Figure 4, we have a collection of rules for definitional equivalence relations between annotations and effects. These rules allow us, when necessary, to treat the $\cup$-constructor that appears in annotations and effects as a commutative, associative, and idempotent operation with $\{\}$ as unit, and to consider annotations and effects as equal up to beta-equivalence and distribution of union over flow construction.

5.2 Type and Effect System

The actual type and effect system is defined in terms of rules for deriving judgements of the form

Annotation sorting $\boxed{\Sigma \vdash \psi :: s}$

$$\frac{\Sigma(\beta) = s}{\Sigma \vdash \beta :: s}\ [sa\text{-}var] \qquad \frac{}{\Sigma \vdash \{\,\} :: \mathtt{ann}}\ [sa\text{-}nil]$$

$$\frac{}{\Sigma \vdash \{\ell\} :: \mathtt{ann}}\ [sa\text{-}sing]$$

$$\frac{\Sigma[\beta \mapsto s_1] \vdash \psi_1 :: s_2}{\Sigma \vdash \lambda\beta :: s_1.\, \psi_1 :: s_1 \to s_2}\ [sa\text{-}abs]$$

$$\frac{\Sigma \vdash \psi_1 :: s_2 \to s \quad \Sigma \vdash \psi_2 :: s_2}{\Sigma \vdash \psi_1\ \psi_2 :: s}\ [sa\text{-}app]$$

$$\frac{\Sigma \vdash \psi_1 :: \mathtt{ann} \quad \Sigma \vdash \psi_2 :: \mathtt{ann}}{\Sigma \vdash \psi_1 \cup \psi_2 :: \mathtt{ann}}\ [sa\text{-}union]$$

Effect sorting $\boxed{\Sigma \vdash \varphi :: s}$

$$\frac{\Sigma(\delta) = s}{\Sigma \vdash \delta :: s}\ [se\text{-}var] \qquad \frac{}{\Sigma \vdash \{\,\} :: \mathtt{eff}}\ [se\text{-}nil]$$

$$\frac{\Sigma \vdash \psi :: \mathtt{ann}}{\Sigma \vdash \{(\ell, \psi)\} :: \mathtt{eff}}\ [se\text{-}sing]$$

$$\frac{\Sigma[\beta \mapsto s_1] \vdash \varphi_1 :: s_2}{\Sigma \vdash \lambda\beta :: s_1.\, \varphi_1 :: s_1 \to s_2}\ [se\text{-}abs\text{-}ann]$$

$$\frac{\Sigma \vdash \varphi_1 :: s_2 \to s \quad \Sigma \vdash \psi :: s_2}{\Sigma \vdash \varphi_1\ \psi :: s}\ [se\text{-}app\text{-}ann]$$

$$\frac{\Sigma[\delta \mapsto s_1] \vdash \varphi_1 :: s_2}{\Sigma \vdash \lambda\delta :: s_1.\, \varphi_1 :: s_1 \to s_2}\ [se\text{-}abs\text{-}eff]$$

$$\frac{\Sigma \vdash \varphi_1 :: s_2 \to s \quad \Sigma \vdash \varphi_2 :: s_2}{\Sigma \vdash \varphi_1\ \varphi_2 :: s}\ [se\text{-}app\text{-}eff]$$

$$\frac{\Sigma \vdash \varphi_1 :: \mathtt{eff} \quad \Sigma \vdash \varphi_2 :: \mathtt{eff}}{\Sigma \vdash \varphi_1 \cup \varphi_2 :: \mathtt{eff}}\ [se\text{-}union]$$

Figure 3. Sorting for annotations and effects.

$$\Sigma \mid \widehat{\Gamma} \vdash t : \widehat{\tau}^{\psi} \,\&\, \varphi,$$

expressing that in the sort environment Σ and the annotated type environment $\widehat{\Gamma}$, the term t can be assigned the annotated type $\widehat{\tau}$ as well as the annotation ψ and the effect φ.

Annotated types are given by

$$\widehat{\tau} \quad \in \quad \widehat{\mathbf{Ty}} \quad \text{annotated types}$$

with

$$\widehat{\tau} \quad ::= \quad \mathtt{bool} \mid \widehat{\tau}_1{}^{\psi_1} \xrightarrow{\varphi} \widehat{\tau}_2{}^{\psi_2} \mid \forall\beta :: s.\, \widehat{\tau}_1 \mid \forall\delta :: s.\, \widehat{\tau}_1.$$

Types are considered equal up to alpha-renaming. We require the argument and result annotations ψ_1 and ψ_2 and the latent effect φ in an annotated function type $\widehat{\tau}_1{}^{\psi_1} \xrightarrow{\varphi} \widehat{\tau}_2{}^{\psi_2}$ to be proper annotations and effects; this requirement is captured by the rules for type well-formedness, listed in Figure 5. We write $\lfloor \widehat{\tau} \rfloor$ for the underlying type that is obtained by removing all annotations and effects from the annotated type $\widehat{\tau}$. If $\lfloor \widehat{\tau} \rfloor = \tau$, we say that $\widehat{\tau}$ is a *completion* of τ.

Annotated type environments $\widehat{\Gamma}$ map variables to pairs $(\widehat{\tau}, \psi)$ consisting of an annotated type $\widehat{\tau}$ and an annotation ψ:

$$\widehat{\Gamma} \quad \in \quad \widehat{\mathbf{TyEnv}} \quad = \quad \mathbf{Var} \to_{\mathrm{fin}} (\widehat{\mathbf{Ty}} \times \mathbf{Ann}) \quad \text{annotated type env.}$$

We write $\lfloor \widehat{\Gamma} \rfloor$ for the underlying type environment that is obtained by removing all annotations and effects from the annotated type environment $\widehat{\Gamma}$.

The rules for flow typing are given in Figure 6. The rule [*f-var*] expresses that the annotated type $\widehat{\tau}$ and the annotation ψ for a

Annotation equivalence $\boxed{\psi \equiv \psi'}$

$$\frac{}{\psi \equiv \psi}\ [qa\text{-}refl] \quad \frac{\psi' \equiv \psi}{\psi \equiv \psi'}\ [qa\text{-}symm] \quad \frac{\psi \equiv \psi'' \quad \psi'' \equiv \psi'}{\psi \equiv \psi'}\ [qa\text{-}trans]$$

$$\frac{\psi_1 \equiv \psi_1'}{\lambda\beta :: s.\, \psi_1 \equiv \lambda\beta :: s.\, \psi_1'}\ [qa\text{-}abs] \qquad \frac{\psi_1 \equiv \psi_1' \quad \psi_2 \equiv \psi_2'}{\psi_1\ \psi_2 \equiv \psi_1'\ \psi_2'}\ [qa\text{-}app]$$

$$\frac{\psi_1 \equiv \psi_1' \quad \psi_2 \equiv \psi_2'}{\psi_1 \cup \psi_2 \equiv \psi_1' \cup \psi_2'}\ [qa\text{-}union]$$

$$\frac{}{(\lambda\beta.\, \psi_{11})\ \psi_2 \equiv [\beta \mapsto \psi_2]\psi_{11}}\ [qa\text{-}beta]$$

$$\frac{}{\psi \equiv \psi \cup \{\,\}}\ [qa\text{-}unit] \qquad \frac{}{\psi \equiv \psi \cup \psi}\ [qa\text{-}idem]$$

$$\frac{}{\psi_1 \cup \psi_2 \equiv \psi_2 \cup \psi_1}\ [qa\text{-}comm]$$

$$\frac{}{\psi_1 \cup (\psi_2 \cup \psi_3) \equiv (\psi_1 \cup \psi_2) \cup \psi_3}\ [qa\text{-}ass]$$

Effect equivalence $\boxed{\varphi \equiv \varphi'}$

$$\frac{}{\varphi \equiv \varphi}\ [qe\text{-}refl] \quad \frac{\varphi' \equiv \varphi}{\varphi \equiv \varphi'}\ [qe\text{-}symm] \quad \frac{\varphi \equiv \varphi'' \quad \varphi'' \equiv \varphi'}{\varphi \equiv \varphi'}\ [qe\text{-}trans]$$

$$\frac{\psi \equiv \psi'}{\{(\ell, \psi)\} \equiv \{(\ell, \psi')\}}\ [qe\text{-}sing]$$

$$\frac{\varphi_1 \equiv \varphi_1'}{\lambda\beta :: s.\, \varphi_1 \equiv \lambda\beta.\, \varphi_1'}\ [qe\text{-}abs\text{-}ann] \qquad \frac{\varphi_1 \equiv \varphi_1' \quad \psi \equiv \psi'}{\varphi_1\ \psi \equiv \varphi_1'\ \psi'}\ [qe\text{-}app\text{-}ann]$$

$$\frac{\varphi_1 \equiv \varphi_1'}{\lambda\delta :: s.\, \varphi_1 \equiv \lambda\delta :: s.\, \varphi_1'}\ [qe\text{-}abs\text{-}eff]$$

$$\frac{\varphi_1 \equiv \varphi_1' \quad \varphi_2 \equiv \varphi_2'}{\varphi_1\ \varphi_2 \equiv \varphi_1'\ \varphi_2'}\ [qe\text{-}app\text{-}eff]$$

$$\frac{\varphi_1 \equiv \varphi_1' \quad \varphi_2 \equiv \varphi_2'}{\varphi_1 \cup \varphi_2 \equiv \varphi_1' \cup \varphi_2'}\ [qe\text{-}union]$$

$$\frac{}{(\lambda\beta.\, \varphi_{11})\ \psi \equiv [\beta \mapsto \psi]\varphi_{11}}\ [qe\text{-}beta\text{-}ann]$$

$$\frac{}{(\lambda\delta.\, \varphi_{11})\ \varphi_2 \equiv [\delta \mapsto \varphi_2]\varphi_{11}}\ [qe\text{-}beta\text{-}eff]$$

$$\frac{}{\varphi \equiv \varphi \cup \{\,\}}\ [qe\text{-}unit] \qquad \frac{}{\varphi \equiv \varphi \cup \varphi}\ [qe\text{-}idem]$$

$$\frac{}{\varphi_1 \cup \varphi_2 \equiv \varphi_2 \cup \varphi_1}\ [qe\text{-}comm]$$

$$\frac{}{\varphi_1 \cup (\varphi_2 \cup \varphi_3) \equiv (\varphi_1 \cup \varphi_2) \cup \varphi_3}\ [qe\text{-}ass]$$

$$\frac{}{\{(\ell, \psi_1) \cup \{(\ell, \psi_2)\} \equiv \{(\ell, \psi_1 \cup \psi_2)\}}\ [qe\text{-}dist]$$

Figure 4. Definitional equivalence for annotations and effects.

Well-formedness $\boxed{\Sigma \vdash \widehat{\tau}\ \mathrm{wft}}$

$$\frac{}{\Sigma \vdash \mathtt{bool}\ \mathrm{wft}}\ [w\text{-}bool]$$

$$\frac{\Sigma \vdash \varphi :: \mathtt{eff} \quad \Sigma \vdash \widehat{\tau}_1\ \mathrm{wft} \quad \Sigma \vdash \psi_1 :: \mathtt{ann} \quad \Sigma \vdash \widehat{\tau}_2\ \mathrm{wft} \quad \Sigma \vdash \psi_2 :: \mathtt{ann}}{\Sigma \vdash \widehat{\tau}_1{}^{\psi_1} \xrightarrow{\varphi} \widehat{\tau}_2{}^{\psi_2}\ \mathrm{wft}}\ [w\text{-}arr]$$

$$\frac{\Sigma[\beta \mapsto s] \vdash \widehat{\tau}_1\ \mathrm{wft}}{\Sigma \vdash \forall\beta :: s.\, \widehat{\tau}_1\ \mathrm{wft}}\ [w\text{-}forall\text{-}ann] \qquad \frac{\Sigma[\delta \mapsto s] \vdash \widehat{\tau}_1\ \mathrm{wft}}{\Sigma \vdash \forall\delta :: s.\, \widehat{\tau}_1\ \mathrm{wft}}\ [w\text{-}forall\text{-}eff]$$

Figure 5. Type well-formedness.

Flow analysis $\boxed{\Sigma \mid \widehat{\Gamma} \vdash t : \widehat{\tau}^{\psi} \,\&\, \varphi}$

$$\frac{\widehat{\Gamma}(x) = (\widehat{\tau}, \psi)}{\Sigma \mid \widehat{\Gamma} \vdash x : \widehat{\tau}^{\psi} \,\&\, \{\,\}}\ [\textit{f-var}] \qquad \frac{}{\Sigma \mid \widehat{\Gamma} \vdash \mathtt{false}^{\ell} : \mathtt{bool}^{\{\ell\}} \,\&\, \{\,\}}\ [\textit{f-false}] \qquad \frac{}{\Sigma \mid \widehat{\Gamma} \vdash \mathtt{true}^{\ell} : \mathtt{bool}^{\{\ell\}} \,\&\, \{\,\}}\ [\textit{f-true}]$$

$$\frac{\Sigma \vdash \widehat{\tau}_1 \text{ wft} \quad \Sigma \vdash \psi_1 :: \mathtt{ann} \quad \Sigma \mid \widehat{\Gamma}[x \mapsto (\widehat{\tau}_1, \psi_1)] \vdash t_1 : \widehat{\tau}_2^{\psi_2} \,\&\, \varphi_0}{\Sigma \mid \widehat{\Gamma} \vdash (\lambda x : \lfloor \widehat{\tau}_1 \rfloor . t_1)^{\ell} : (\widehat{\tau}_1^{\psi_1} \xrightarrow{\varphi_0} \widehat{\tau}_2^{\psi_2})^{\{\ell\}} \,\&\, \{\,\}}\ [\textit{f-abs}]$$

$$\frac{\Sigma \mid \widehat{\Gamma} \vdash t_1 : \mathtt{bool}^{\psi_1} \,\&\, \varphi_1 \quad \Sigma \mid \widehat{\Gamma} \vdash t_2 : \widehat{\tau}^{\psi} \,\&\, \varphi_2 \quad \Sigma \mid \widehat{\Gamma} \vdash t_3 : \widehat{\tau}^{\psi} \,\&\, \varphi_3}{\Sigma \mid \widehat{\Gamma} \vdash (\textbf{if } t_1 \textbf{ then } t_2 \textbf{ else } t_3)^{\ell} : \widehat{\tau}^{\psi} \,\&\, \varphi_1 \cup \{(\ell, \psi_1)\} \cup \varphi_2 \cup \varphi_3}\ [\textit{f-if}]$$

$$\frac{\Sigma \mid \widehat{\Gamma} \vdash t_1 : (\widehat{\tau}_2^{\psi_2} \xrightarrow{\varphi_0} \widehat{\tau}^{\psi})^{\psi_1} \,\&\, \varphi_1 \quad \Sigma \mid \widehat{\Gamma} \vdash t_2 : \widehat{\tau}_2^{\psi_2} \,\&\, \varphi_2}{\Sigma \mid \widehat{\Gamma} \vdash (t_1\ t_2)^{\ell} : \widehat{\tau}^{\psi} \,\&\, \varphi_1 \cup \varphi_2 \cup \{(\ell, \psi_1)\} \cup \varphi_0}\ [\textit{f-app}] \qquad \frac{\Sigma \mid \widehat{\Gamma} \vdash t_1 : (\widehat{\tau}^{\psi} \xrightarrow{\varphi_0} \widehat{\tau}^{\psi})^{\psi_1} \,\&\, \varphi_1}{\Sigma \mid \widehat{\Gamma} \vdash (\textbf{fix}\ t_1)^{\ell} : \widehat{\tau}^{\psi} \,\&\, \varphi_1 \cup \{(\ell, \psi_1)\} \cup \varphi_0}\ [\textit{f-fix}]$$

$$\frac{\Sigma[\beta \mapsto s] \mid \widehat{\Gamma} \vdash t : \widehat{\tau}_1^{\psi} \,\&\, \varphi}{\Sigma \mid \widehat{\Gamma} \vdash t : (\forall \beta :: s.\, \widehat{\tau}_1)^{\psi} \,\&\, \varphi}\ [\textit{f-gen-ann}] \qquad \frac{\Sigma \mid \widehat{\Gamma} \vdash t : (\forall \beta :: s.\, \widehat{\tau}_1)^{\psi} \,\&\, \varphi \quad \Sigma \vdash \psi_0 :: s}{\Sigma \mid \widehat{\Gamma} \vdash t : ([\beta \mapsto \psi_0]\widehat{\tau}_1)^{\psi} \,\&\, \varphi}\ [\textit{f-inst-ann}]$$

$$\frac{\Sigma[\delta \mapsto s] \mid \widehat{\Gamma} \vdash t : \widehat{\tau}_1^{\psi} \,\&\, \varphi}{\Sigma \mid \widehat{\Gamma} \vdash t : (\forall \delta :: s.\, \widehat{\tau}_1)^{\psi} \,\&\, \varphi}\ [\textit{f-gen-eff}] \qquad \frac{\Sigma \mid \widehat{\Gamma} \vdash t : (\forall \delta :: s.\, \widehat{\tau}_1)^{\psi} \,\&\, \varphi \quad \Sigma \vdash \varphi_0 :: s}{\Sigma \mid \widehat{\Gamma} \vdash t : ([\delta \mapsto \varphi_0]\widehat{\tau}_1)^{\psi} \,\&\, \varphi}\ [\textit{f-inst-eff}]$$

$$\frac{\begin{array}{c}\Sigma \mid \widehat{\Gamma} \vdash t : \widehat{\tau}^{\psi'} \,\&\, \varphi' \\ \psi \equiv \psi' \quad \Sigma \vdash \psi :: \mathtt{ann} \quad \varphi \equiv \varphi' \quad \Sigma \vdash \varphi :: \mathtt{eff}\end{array}}{\Sigma \mid \widehat{\Gamma} \vdash t : \widehat{\tau}^{\psi} \,\&\, \varphi}\ [\textit{f-eq}] \qquad \frac{\begin{array}{c}\Sigma \mid \widehat{\Gamma} \vdash t : \widehat{\tau}'^{\psi_1} \,\&\, \varphi_1 \\ \widehat{\tau}' \leqslant \widehat{\tau} \quad \Sigma \vdash \widehat{\tau} \text{ wft} \quad \Sigma \vdash \psi_2 :: \mathtt{ann} \quad \Sigma \vdash \varphi_2 :: \mathtt{eff}\end{array}}{\Sigma \mid \widehat{\Gamma} \vdash t : \widehat{\tau}^{(\psi_1 \cup \psi_2)} \,\&\, (\varphi_1 \cup \varphi_2)}\ [\textit{f-sub}]$$

Subtyping $\boxed{\widehat{\tau} \leqslant \widehat{\tau}'}$

$$\frac{}{\widehat{\tau} \leqslant \widehat{\tau}'}\ [\textit{s-refl}] \qquad \frac{\varphi' \equiv \varphi \cup \varphi'' \quad \widehat{\tau}_1' \leqslant \widehat{\tau}_1 \quad \psi_1 \equiv \psi_1' \cup \psi_1'' \quad \widehat{\tau}_2 \leqslant \widehat{\tau}_2' \quad \psi_2' \equiv \psi_2 \cup \psi_2''}{\widehat{\tau}_1^{\psi_1} \xrightarrow{\varphi} \widehat{\tau}_2^{\psi_2} \leqslant \widehat{\tau}_1'^{\psi_1'} \xrightarrow{\varphi'} \widehat{\tau}_2'^{\psi_2'}}\ [\textit{s-arr}]$$

$$\frac{\widehat{\tau}_1 \leqslant \widehat{\tau}_1'}{\forall \beta :: s.\, \widehat{\tau}_1 \leqslant \forall \beta :: s.\, \widehat{\tau}_1'}\ [\textit{s-forall-ann}] \qquad \frac{\widehat{\tau}_1 \leqslant \widehat{\tau}_1'}{\forall \delta :: s.\, \widehat{\tau}_1 \leqslant \forall \delta :: s.\, \widehat{\tau}_1'}\ [\textit{s-forall-eff}]$$

Figure 6. Type and effect system for flow analysis.

variable x are to be retrieved from the annotated type environment $\widehat{\Gamma}$. In the call-by-value semantics of our language, the evaluation of a variable does not result in flow; hence, the effect component in the conclusion of rule [*f-var*] stays empty. For the Boolean producers $\mathtt{false}^{\ell}$ and $\mathtt{true}^{\ell}$ we have axioms [*f-false*] and [*f-true*] that assign the annotated type $\mathtt{bool}$ and a singleton annotation $\{\ell\}$ that reflects the production site ℓ. Producers are already fully evaluated and so no effect is recorded.

Lambda-abstractions $(\lambda x : \tau . t_1)^{\ell}$ are dealt with by the rule [*f-abs*]. It states that the body t_1 of the abstraction is to be analysed in an extended annotated type environment that maps the formal parameter x to the pair $(\widehat{\tau}_1, \psi_1)$, where ψ_1 is a proper annotation and $\widehat{\tau}_1$ a possibly polymorphic completion of τ that is well-formed with respect to the sorting environment Σ. While $\widehat{\tau}_1$ and ψ_1 are then used as the argument type and annotation for the abstraction, the annotated type $\widehat{\tau}_2$ and the annotation ψ_2, obtained from the analysis of the body, both end up in result position; the effect φ_0 of t_1 constitutes the latent effect. The annotation and effect for the abstraction as a whole are taken to be $\{\ell\}$ and $\{\,\}$, respectively.

The rule for conditionals $(\textbf{if } t_1 \textbf{ then } t_2 \textbf{ else } t_3)^{\ell}$, [*f-if*], requires the condition t_1 to be of Boolean type and the branches t_2 and t_3 to agree on their annotated types and annotations, which will then be used as the annotated type and annotation for the conditional itself. The effect for the conditional is constructed by taking the union over the effects of the three subterms and recording that the Boolean values that may flow to the condition t_1 are possibly consumed at the site labelled with ℓ.

In the rule [*f-app*] for applications $(t_1\ t_2)^{\ell}$, the annotated type $\widehat{\tau}_2$ and the annotation ψ_2 of the argument term t_2 are to match with the argument type and annotation of the function term t_1. The annotated type $\widehat{\tau}$ and annotation ψ are then retrieved from the result positions in the type of t_1. The effect for the application subsumes the effects for its subterms t_1 and t_2 as well as the possible flow from the function labels ψ_1 to the application site ℓ and the latent effect φ_0 of t_1.

For the fixed point $(\textbf{fix}\ t_1)^{\ell}$ of a term t_1, the annotated type $\widehat{\tau}^{\psi}$ is retrieved from the type of t_1, which is required to be of the form $\tau^{\psi} \xrightarrow{\varphi_0} \tau^{\psi}$. The effect component is then constructed by combining the effect φ_1 of t_1, the singleton effect $\{(\ell, \psi_1)\}$ with ψ_1 the annotation of t_1, and the latent effect φ_0 of t_1.

The rules [*f-gen-ann*] and [*f-inst-ann*] form a pair of introduction and elimination rules for annotation polymorphism. Quantification over an s-sorted annotation is allowed, if the corresponding binding in the sort environment admits a valid analysis. Instantiation requires an annotation of appropriate sort to be supplied. Rules [*f-gen-eff*] and [*f-inst-eff*] are analogue rules for effect polymorphism. The rule [*f-eq*] expresses that annotations and effects at top level can always be safely replaced by well-sorted definitional equivalents.

The rule [*f-sub*], finally, is a combined rule for subtyping and subeffecting (Tang and Jouvelot 1995) that allows for overapproximation of annotations and effects. This rule is typically used immediately before the rule [*f-if*] in order to have the branches of a conditional agree on their types and annotations. The rules for subtyping are given in the lower part of Figure 6.

6. Properties

Let us now briefly review the most important metatheoretical properties of our type and effect system.

6.1 Applicability

Our flow analysis is a conservative extension of the underlying type system from Section 3 in the sense that every program typeable in the underlying system can be successfully subjected to the analysis. Furthermore, both systems agree on the shape of types assignable.

Theorem 1 (Conservative extension).

1. If $\Gamma \vdash t : \tau$, then there exist $\widehat{\Gamma}$, $\widehat{\tau}$, ψ, and φ with $\lfloor\widehat{\Gamma}\rfloor = \Gamma$ and $\lfloor\widehat{\tau}\rfloor = \tau$, such that $[\,] \mid \Gamma \vdash t : \widehat{\tau}^{\psi} \,\&\, \varphi$.
2. If $\Sigma \mid \widehat{\Gamma} \vdash t : \widehat{\tau}^{\psi} \,\&\, \varphi$, then $\lfloor\widehat{\Gamma}\rfloor \vdash t : \lfloor\widehat{\tau}\rfloor$. ∎

6.2 Semantic Correctness

To establish the correctness of the analysis with respect to the instrumented natural semantics from Section 3, we consider interpretations $[\![\cdot]\!]$ of annotations ψ as sets of labels,

$$\begin{array}{ll} [\![\{\,\}]\!] & = \{\,\} \\ [\![\{\ell\}]\!] & = \{\ell\} \\ [\![\psi_1 \cup \psi_2]\!] & = [\![\psi_1]\!] \cup [\![\psi_2]\!], \end{array}$$

and of effects φ as flows,

$$\begin{array}{ll} [\![\{\,\}]\!] & = \{\,\} \\ [\![\{\ell, \psi\}]\!] & = \{(\ell, \ell') \mid \ell' \in [\![\psi]\!]\} \\ [\![\varphi_1 \cup \varphi_2]\!] & = [\![\varphi_1]\!] \cup [\![\varphi_2]\!]. \end{array}$$

Both interpretations are partial in the sense that they do not account for abstractions, applications, and free variables in annotations and effects. Hence, we only consider closed environments and observe that the type and effect system guarantees all top-level annotations and effects to be proper annotations and effects.

Lemma 2. If $[\,] \mid [\,] \vdash t : \widehat{\tau}^{\psi} \,\&\, \varphi$, then $[\,] \vdash \widehat{\tau}$ wft, $[\,] \vdash \psi :: \mathtt{ann}$, and $[\,] \vdash \varphi :: \mathtt{eff}$. ∎

As proper annotations and effects are always definitionally equivalent to forms without abstractions and applications, we can now formulate the following result.

Theorem 3 (Semantic soundness). If $[\,] \mid [\,] \vdash t : \widehat{\tau}^{\psi} \,\&\, \varphi$ and $t \Downarrow_{\mathsf{F}} p^{\ell}$, then there exist ψ' and φ' with $\psi \equiv \psi'$ and $\varphi \equiv \varphi'$, such that $\ell \in [\![\psi']\!]$ and $\mathsf{F} \subseteq [\![\varphi']\!]$. ∎

6.3 Existence of "Best" Analyses

While Theorem 1 establishes that all well-typed programs can be analysed, we now wish to state that each analysable program admits an analysis that is in some sense "better" than all other analyses for that program. As we are interested in analyses that guarantee modularity, we shall restrict ourselves to analyses that provide fully flexible types.

To this end, let χ range over both annotation and effect variables, together referred to as *flow variables*,

$$\chi \quad \in \quad \mathbf{AnnVar} \cup \mathbf{EffVar} \quad \text{flow variables},$$

and let us use overbar notation to denote sequences, where we feel free to "downcast" sequences of flow variables to sets of flow variables. We write ε for the empty sequence, $\mathit{ffv}(\widehat{\tau})$ and $\mathit{ffv}(\widehat{\Gamma})$ for the set of free, i.e., unbound, flow variables in, respectively, an annotated type $\widehat{\tau}$ and an annotated type environment $\widehat{\Gamma}$, and $\mathit{annvars}(\overline{\chi_i})$ for the subsequence of annotation variables contained in $\overline{\chi_i}$. Then, fully flexible types are defined as follows.

Definition 1. An annotated type $\widehat{\tau}$ is *fully parametric* if

1. $\widehat{\tau} = \mathtt{bool}$, or
2. $\widehat{\tau} = (\overline{\forall \chi_i :: s_i.\, \widehat{\tau}_1}^{\beta} \xrightarrow{\delta_0 \overline{\chi_i}} \widehat{\tau}_2^{(\beta_0 \overline{\beta_{i'}})})$ for some δ_0 and β_0 with (a) $\widehat{\tau}_1$ and $\widehat{\tau}_2$ fully parametric, (b) $\overline{\chi_i} = \{\beta\} \cup \mathit{ffv}(\widehat{\tau}_1)$, and (c) $\overline{\beta_{i'}} = \mathit{annvars}(\overline{\chi_i})$. ∎

Definition 2. An annotated type $\widehat{\tau}$ is *fully flexible* if

1. $\widehat{\tau} = \mathtt{bool}$, or
2. $\widehat{\tau} = (\overline{\forall \chi_i :: s_i.\, \widehat{\tau}_1}^{\beta} \xrightarrow{\varphi} \widehat{\tau}_2^{\psi_2})$ for some φ and ψ_2 with (a) $\widehat{\tau}_1$ fully parametric, (b) $\widehat{\tau}_2$ fully flexible, and (c) $\overline{\chi_i} = \{\beta\} \cup \mathit{ffv}(\widehat{\tau}_1)$. ∎

Note that full parametricity implies full flexibility and how higher-order function types give rise to higher-ranked polymorphism and higher-order operators over annotations and effects. Full flexibility extends naturally to closed type environments.

Definition 3. An annotated type environment $\widehat{\Gamma}$ is *fully flexible* if $\mathit{ffv}(\widehat{\Gamma}) = \{\,\}$ and if, for all x, $\widehat{\tau}$, and ψ with $\widehat{\Gamma}(x) = (\widehat{\tau}, \psi)$, we have that $\widehat{\tau}$ is fully flexible. ∎

Now, in a fully flexible environment, each analysable term admits a fully flexible type.

Lemma 4. If $[\,] \mid \widehat{\Gamma} \vdash t : \widehat{\tau}'^{\psi'} \,\&\, \varphi'$ with $\widehat{\Gamma}$ fully flexible, then there exist $\widehat{\tau}$, ψ, and φ such that $\widehat{\tau}$ is fully flexible and $\widehat{\Gamma} \vdash t : \widehat{\tau}^{\psi} \,\&\, \varphi$. ∎

Amongst all possible analyses for a given term in a given environment, we are interested in a fully flexible analysis that makes the most accurate prediction about production sites and flow, i.e., the analysis that results in the "smallest" types, annotations, and effects. As all fully flexible types for a term agree on their negative positions, the notion of a best analysis can be straightforwardly expressed in terms of subtyping and definitional equivalence.

Definition 4. The triple $(\widehat{\tau}, \psi, \varphi)$ consisting of a fully flexible annotated type $\widehat{\tau}$, an annotation ψ, and an effect φ constitutes a *best analysis* for t in $\widehat{\Gamma}$, if $[\,] \mid \widehat{\Gamma} \vdash t : \widehat{\tau}^{\psi} \,\&\, \varphi$ and if, for all $\widehat{\tau}'$, ψ', and φ' with $[\,] \mid \widehat{\Gamma} \vdash t : \widehat{\tau}'^{\psi'} \,\&\, \varphi'$ and $\widehat{\tau}'$ fully flexible, we have that $\widehat{\tau} \leqslant \widehat{\tau}'$, $\psi' \equiv \psi \cup \psi''$, and $\varphi' \equiv \varphi \cup \varphi''$ for some ψ'' and φ''. ∎

Theorem 5 (Existence of best analyses). If $[\,] \mid \widehat{\Gamma} \vdash t : \widehat{\tau}'^{\psi'} \,\&\, \varphi'$ with $\widehat{\tau}'$ fully flexible, then there exist $\widehat{\tau}$, ψ, and φ, such that $(\widehat{\tau}, \psi, \varphi)$ is a best analysis for t in $\widehat{\Gamma}$. ∎

7. Algorithm

In this section, we present an inference algorithm for obtaining best analyses. The algorithm naturally breaks up in two parts: a reconstruction algorithm $\mathbf{R}$ that produces annotated types, annotations, and effects for terms as well as constraints between flow variables (Section 7.1), and a procedure $\mathbf{S}$ for solving the constraints produced by $\mathbf{R}$ (Section 7.2).

A crucial aspect of the algorithm is that the constraints that are generated for the body of a lambda-abstraction are solved locally, allowing for annotations and effects to be generalised over at the binding-sites of formal parameters.

7.1 Flow Reconstruction

The algorithm $\mathbf{R}$ for reconstructing types, annotations, and effects is given in Figure 7. It takes as input a pair $(\widehat{\Gamma}, t)$ consisting of an annotated type environment $\widehat{\Gamma}$ and a term t and produces a quadruple $(\widehat{\tau}, \beta, \delta, C)$ consisting of an annotated type $\widehat{\tau}$, an annotation variable β, an effect variable δ, and a finite set C of constraints over β and δ as well as any intermediate flow variables. Constraints are given by

$$\begin{array}{rcll} q & \in & \mathbf{Constraint} & \text{constraints} \\ C & \in & \mathscr{F}(\mathbf{Constraint}) & \text{constraint sets,} \end{array}$$

where

$$q \quad ::= \quad \psi \subseteq \beta \;\mid\; \varphi \subseteq \delta.$$

That is, a constraint expresses either the inclusion of an annotation ψ in the annotation represented by the annotation variable β or

$$\mathbf{R}(\widehat{\Gamma},x) = \text{let } (\widehat{\tau},\psi) = \widehat{\Gamma}(x)$$
$$\quad \beta,\delta \text{ be fresh}$$
$$\quad \text{in } (\widehat{\tau},\beta,\delta,\{\,\psi \subseteq \beta\,\})$$

$$\mathbf{R}(\widehat{\Gamma},\mathtt{false}^{\ell}) = \text{let } \beta,\delta \text{ be fresh in } (\mathtt{bool},\beta,\delta,\{\{\ell\} \subseteq \beta\,\})$$

$$\mathbf{R}(\widehat{\Gamma},\mathtt{true}^{\ell}) = \text{let } \beta,\delta \text{ be fresh in } (\mathtt{bool},\beta,\delta,\{\{\ell\} \subseteq \beta\,\})$$

$$\mathbf{R}(\widehat{\Gamma},(\lambda x:\tau_1.t_1)^{\ell}) =$$
$$\quad \text{let } (\widehat{\tau}_1,\overline{\chi_i :: s_i}) = \mathbf{C}(\tau_1,\varepsilon)$$
$$\qquad \beta_1 \text{ be fresh}$$
$$\qquad (\widehat{\tau}_2,\beta_2,\delta_0,C_1) = \mathbf{R}(\widehat{\Gamma}[x \mapsto (\widehat{\tau}_1,\beta_1)],t_1)$$
$$\qquad X = \{\,\beta_1\,\} \cup \overline{\{\,\chi_i\,\}} \cup \mathit{ffv}(\widehat{\Gamma})$$
$$\qquad (\psi_2,\varphi_0) = \mathbf{S}(C_1,X,\beta_2,\delta_0)$$
$$\qquad \widehat{\tau} = \forall \beta_1 :: \mathtt{ann}.\overline{\forall \chi_i :: s_i}.\widehat{\tau}_1^{\beta_1} \xrightarrow{\varphi_0} \widehat{\tau}_2^{\psi_2}$$
$$\qquad \beta,\delta \text{ be fresh}$$
$$\quad \text{in } (\widehat{\tau},\beta,\delta,\{\{\ell\} \subseteq \beta\,\})$$

$$\mathbf{R}(\widehat{\Gamma},(\mathbf{if}\ t_1\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3)^{\ell}) =$$
$$\quad \text{let } (\mathtt{bool},\beta_1,\delta_1,C_1) = \mathbf{R}(\widehat{\Gamma},t_1)$$
$$\qquad (\widehat{\tau}_2,\beta_2,\delta_2,C_2) = \mathbf{R}(\widehat{\Gamma},t_2)$$
$$\qquad (\widehat{\tau}_3,\beta_3,\delta_3,C_3) = \mathbf{R}(\widehat{\Gamma},t_3)$$
$$\qquad \widehat{\tau} = \mathbf{J}(\widehat{\tau}_2,\widehat{\tau}_3)$$
$$\qquad \beta,\delta \text{ be fresh}$$
$$\qquad C = \{\,\delta_1 \subseteq \delta\,\} \cup \{\{(\ell,\beta_1)\} \subseteq \delta\,\} \cup \{\,\delta_2 \subseteq \delta\,\} \cup \{\,\delta_3 \subseteq \delta\,\} \cup$$
$$\qquad\quad \{\,\beta_2 \subseteq \beta\,\} \cup \{\,\beta_3 \subseteq \beta\,\} \cup C_1 \cup C_2 \cup C_3$$
$$\quad \text{in } (\widehat{\tau},\beta,\delta,C)$$

$$\mathbf{R}(\widehat{\Gamma},(t_1\ t_2)^{\ell}) =$$
$$\quad \text{let } (\widehat{\tau}_1,\beta_1,\delta_1,C_1) = \mathbf{R}(\widehat{\Gamma},t_1)$$
$$\qquad (\widehat{\tau}_2,\beta_2,\delta_2,C_2) = \mathbf{R}(\widehat{\Gamma},t_2)$$
$$\qquad \widehat{\tau}_2'^{\beta_2'} \xrightarrow{\varphi_0'} \widehat{\tau}'^{\psi'} = \mathbf{I}(\widehat{\tau}_1)$$
$$\qquad \theta = [\beta_2' \mapsto \beta_2] \circ \mathbf{M}([\,],\widehat{\tau}_2,\widehat{\tau}_2')$$
$$\qquad \beta,\delta \text{ be fresh}$$
$$\qquad C = \{\,\delta_1 \subseteq \delta\,\} \cup \{\,\delta_2 \subseteq \delta\,\} \cup \{\{(\ell,\beta_1)\} \subseteq \delta\,\} \cup \{\,\theta\ \varphi_0' \subseteq \delta\,\} \cup$$
$$\qquad\quad \{\,\theta\ \psi' \subseteq \beta\,\} \cup C_1 \cup C_2$$
$$\quad \text{in } (\theta\ \widehat{\tau}',\beta,\delta,C)$$

$$\mathbf{R}(\widehat{\Gamma},(\mathbf{fix}\ t_1)^{\ell}) =$$
$$\quad \text{let } (\widehat{\tau}_1,\beta_1,\delta_1,C_1) = \mathbf{R}(\widehat{\Gamma},t_1)$$
$$\qquad \widehat{\tau}'^{\beta'} \xrightarrow{\varphi_0'} \widehat{\tau}''^{\psi''} = \mathbf{I}(\widehat{\tau}_1)$$
$$\qquad \theta_1 = \mathbf{M}([\,],\widehat{\tau}'',\widehat{\tau}')$$
$$\qquad \theta_2 = [\beta' \mapsto \theta_1\ \psi'']$$
$$\qquad \beta,\delta \text{ be fresh}$$
$$\qquad C = \{\,\delta_1 \subseteq \delta\,\} \cup \{\{(\ell,\beta_1)\} \subseteq \delta\,\} \cup \{\,\theta_2\ (\theta_1\ \varphi_0') \subseteq \delta\,\} \cup$$
$$\qquad\quad \{\,\theta_2\ (\theta_1\ \psi'') \subseteq \beta\,\} \cup C_1$$
$$\quad \text{in } (\theta_2\ (\theta_1\ \widehat{\tau}'),\beta,\delta,C)$$

Figure 7. Reconstruction algorithm.

$$\mathbf{C}(\mathtt{bool},\overline{\chi_i :: s_i}) = (\mathtt{bool},\{\,\})$$

$$\mathbf{C}(\tau_1 \to \tau_2,\overline{\chi_i :: s_i}) =$$
$$\quad \text{let } (\widehat{\tau}_1,\overline{\chi_j :: s_j}) = \mathbf{C}(\tau_1,\varepsilon)$$
$$\qquad \beta_1 \text{ be fresh}$$
$$\qquad (\widehat{\tau}_2,\overline{\chi_k :: s_k}) = \mathbf{C}(\tau_2,(\overline{\chi_i :: s_i},\beta_1 :: \mathtt{ann},\overline{\chi_j :: s_j}))$$
$$\qquad \overline{\beta_{i'} :: s_{i'}} = \mathit{annvars}(\overline{\chi_i :: s_i})$$
$$\qquad \overline{\beta_{j'} :: s_{j'}} = \mathit{annvars}(\overline{\chi_j :: s_j})$$
$$\qquad \beta_0,\delta_0 \text{ be fresh}$$
$$\quad \text{in } (\forall \beta_1 :: \mathtt{ann}.\overline{\forall \chi_j :: s_j}.\widehat{\tau}_1^{\beta_1} \xrightarrow{\delta_0 \overline{\chi_i} \beta_1 \overline{\chi_j}} \widehat{\tau}_2^{(\beta_0 \overline{\beta_{i'}} \beta_1 \overline{\beta_{j'}})},$$
$$\qquad (\delta_0 :: \overline{s_i \to} \ \mathtt{ann} \to \overline{s_j \to}\ \mathtt{eff}, \beta_0 :: \overline{s_{i'} \to}\ \mathtt{ann} \to \overline{s_{j'} \to}\ \mathtt{ann},$$
$$\qquad \overline{\chi_k :: s_k}))$$

Figure 8. Completion algorithm.

$$\mathbf{J}(\mathtt{bool},\mathtt{bool}) = \mathtt{bool}$$

$$\mathbf{J}(\widehat{\tau}_1^{\beta_1} \xrightarrow{\varphi_1} \widehat{\tau}_{12}^{\psi_{12}},\widehat{\tau}_1^{\beta_1} \xrightarrow{\varphi_2} \widehat{\tau}_{22}^{\psi_{22}}) = \widehat{\tau}_1^{\beta_1} \xrightarrow{\varphi_1 \cup \varphi_2} \mathbf{J}(\widehat{\tau}_{12},\widehat{\tau}_{22})^{(\psi_{12} \cup \psi_{22})}$$

$$\mathbf{J}(\forall \beta :: s.\,\widehat{\tau}_{11},\forall \beta :: s.\,\widehat{\tau}_{21}) = \forall \beta :: s.\,\mathbf{J}(\widehat{\tau}_{11},\widehat{\tau}_{21})$$

$$\mathbf{J}(\forall \delta :: s.\,\widehat{\tau}_{11},\forall \delta :: s.\,\widehat{\tau}_{21}) = \forall \delta :: s.\,\mathbf{J}(\widehat{\tau}_{11},\widehat{\tau}_{21})$$

$$\mathbf{J}(\widehat{\tau}_1,\widehat{\tau}_2) = \text{fail} \qquad \text{in all other cases}$$

Figure 9. Join algorithm.

the inclusion of an effect φ in the effect represented by the effect variable δ. We carefully maintain the invariant that all annotated types produced are fully flexible.

Turning to the details of the algorithm, the cases for variables and Boolean constants $\mathtt{false}^{\ell}$ and $\mathtt{true}^{\ell}$ are straightforward: we generate fresh annotation and effect variables and propagate the relevant information from either the type environment $\widehat{\Gamma}$ or the producer label ℓ to the result tuple.

More interesting is the case for lambda-abstractions $(\lambda x : \tau_1.t_1)^{\ell}$. Here, we first make a call to the subsidiary procedure $\mathbf{C}$, given in Figure 8, that produces a pair $(\widehat{\tau}_1,\overline{\chi_i :: s_i})$ consisting of a fully parametric (cf. Definition 1) completion $\widehat{\tau}_1$ of τ_1 and a sequence $\overline{\chi_i :: s_i}$ that contains the free flow variables of $\widehat{\tau}_1$ accompanied by their sorts. Then we create a mapping from the formal parameter x to the pair $(\widehat{\tau}_1,\beta_1)$, where β_1 is a fresh annotation variable, and use it in a recursive invocation of $\mathbf{R}$ for the body t_1 of the abstraction. This recursive invocation results in a tuple $(\widehat{\tau}_2,\beta_2,\delta_0,C_1)$. The constraints in C_1 are then solved with respect to a finite set of *active flow variables* X (see Section 7.2),

$$X \quad \in \quad \mathscr{F}(\mathbf{AnnVar} \cup \mathbf{EffVar}) \quad \text{flow-variable sets},$$

to yield a least solution (ψ_2,φ_0) for the flow variables β_2 and δ_0. An annotated type for the abstraction is then formed by quantifying over the argument annotation variable β_1 and the free flow variables $\overline{\chi_i}$ of the argument type $\widehat{\tau}_1$; choosing $\widehat{\tau}_1$ and β_1 as argument type and annotation; choosing $\widehat{\tau}_2$ and ψ_2 as result type and annotation; and, choosing φ_0 as latent effect. For the annotation and effect of the abstraction as a whole, we pick fresh variables β and δ and record that ℓ is to be included in a solution for β.

For conditionals $(\mathbf{if}\ t_1\ \mathbf{then}\ t_2\ \mathbf{else}\ t_3)^{\ell}$ we make recursive calls to $\mathbf{R}$ for all three subterms. The thus obtained constraint sets C_1, C_2, and C_3 are then combined with the constraints that account for the flow that is involved with evaluating a conditional to form the constraint set C for the conditional as a whole. The annotated type $\widehat{\tau}$ for the conditional is obtained by taking the least upper bound of the recursively obtained types $\widehat{\tau}_2$ and $\widehat{\tau}_3$ with respect to the subtyping relation of Figure 6. This least upper bound is computed by the join algorithm $\mathbf{J}$ in Figure 9. Note how $\mathbf{J}$ makes essential use of the invariant that all types are fully flexible (and that the types to join thus agree in their argument positions) as well as the fact that types are to be considered equal up to alpha-renaming (in the cases for quantified types).

In the case for applications $(t_1\ t_2)^{\ell}$, we make recursive calls to $\mathbf{R}$ for the function term t_1 and the argument term t_2. The thus obtained annotated type for $\widehat{\tau}_1$ for t_1, for which our invariant guarantees that it is fully flexible, is then instantiated by means of a call to the auxiliary procedure $\mathbf{I}$ (Figure 10), from which we retrieve the

$$\mathbf{I}(\forall\beta :: s.\,\widehat{\tau}_1) = \text{let } \beta' \text{ be fresh in } [\beta \mapsto \beta'](\mathbf{I}(\widehat{\tau}_1))$$

$$\mathbf{I}(\forall\delta :: s.\,\widehat{\tau}_1) = \text{let } \delta' \text{ be fresh in } [\delta \mapsto \delta'](\mathbf{I}(\widehat{\tau}_1))$$

$$\mathbf{I}(\widehat{\tau}) = \widehat{\tau} \qquad \text{in all other cases}$$

Figure 10. Instantiation algorithm.

$$\mathbf{M}(\Sigma, \texttt{bool}, \texttt{bool}) = id$$

$$\mathbf{M}(\Sigma, \widehat{\tau}_1^{\beta_1} \xrightarrow{\varphi} \widehat{\tau}_2^{\psi_2}, \widehat{\tau}_1^{\beta_1} \xrightarrow{\delta_0\overline{\chi_i}} \widehat{\tau}_2'^{\beta_0\overline{\beta_j}}) = [\delta_0 \mapsto (\overline{\lambda\chi_i :: \Sigma(\chi_i).\ \varphi})] \circ [\beta_0 \mapsto (\overline{\lambda\beta_j :: \Sigma(\beta_j).\ \psi_2})] \circ \mathbf{M}(\Sigma, \widehat{\tau}_2, \widehat{\tau}_2')$$

$$\mathbf{M}(\Sigma, \forall\beta :: s.\,\widehat{\tau}_1, \forall\beta :: s, \widehat{\tau}_1') = \mathbf{M}(\Sigma[\beta \mapsto s], \widehat{\tau}_1, \widehat{\tau}_2)$$

$$\mathbf{M}(\Sigma, \forall\delta :: s.\,\widehat{\tau}_1, \forall\delta :: s, \widehat{\tau}_1') = \mathbf{M}(\Sigma[\delta \mapsto s], \widehat{\tau}_1, \widehat{\tau}_2)$$

$$\mathbf{M}(\Sigma, \widehat{\tau}, \widehat{\tau}') = \text{fail} \qquad \text{in all other cases}$$

Figure 11. Matching algorithm.

fully parametric parameter type $\widehat{\tau}_2'$ and the parameter annotation β_2'. Against these we then match the actual argument type $\widehat{\tau}_2$ and the actual argument annotation β_2, resulting in a substitution θ,

$$\theta \quad \in \quad \textbf{Subst} \quad \text{substitutions.}$$

For the matching of $\widehat{\tau}_2$ against $\widehat{\tau}_2'$ we rely on a subsidiary procedure $\mathbf{M}$, given in Figure 11. The substitution θ is used to determine the annotated type of the application as a whole from the result type $\widehat{\tau}'$ from t_1. For the annotation and the effect of the application, we generate fresh variables β and δ and in the constraint set C we include constraints obtained for t_1 and t_2 as well as the constraints that are obtained by considering the flow incurred by the application.

Finally, the case for fixed points $(\mathbf{fix}\, t_1)^\ell$ is similar to the case for applications with the most important difference that a substitution is constructed in two steps here. First, a substitution θ_1 is constructed by matching the result type of t_1 against its fully parametric parameter type. Then, the "recursive knot is tied" by substituting the result annotation for the annotation variable β' that constitutes the parameter annotation.

7.2 Constraint Solving

For solving the constraints produced by the reconstruction algorithm $\mathbf{R}$, we rely on a standard worklist algorithm. This algorithm, $\mathbf{S}$, is given in Figure 12. As inputs it takes a constraint set C, a set of active flow variables X that are to be considered as constants during solving, an annotation variable β, and an effect variable δ. As outputs it produces least solutions ψ and φ for β and δ under C.

During solving there is no need for explicitly distinguishing between annotation constraints and effect constaints. Therefore we take

$$\xi \quad \in \quad \textbf{Ann} \cup \textbf{Eff} \quad \text{flow terms}$$

and write all constraints as $\xi \subseteq \chi$.

The algorithm maintains a finite set worklist for keeping track of constraints that are still to be considered. Furthermore, it uses a finite map analysis from flow variables to flow terms, in which intermediate solutions for β, δ, and the flow variables in X and the right-hand sides of C are kept; and a finite map dependencies that stores, for each flow variable χ, which constraints need to be reconsidered if the solution for χ is updated.

$\mathbf{S}(C, X, \beta, \delta) =$ do
- (* initialisation *)
- worklist $:= \{\,\}$
- analysis $:= [\,]$
- dependencies $:= [\,]$
- for all $(\xi \subseteq \chi)$ in C do
  - worklist $:=$ worklist $\cup \{\xi \subseteq \chi\}$
  - analysis $:=$ analysis$[\chi \mapsto \{\,\}]$
  - for all ξ' in $ffv(\xi)$ do dependencies $:=$ dependencies$[\xi' \mapsto \{\,\}]$
- for all $(\xi \subseteq \chi)$ in C do
  - for all ξ' in $ffv(\xi)$ do
    - dependencies $:=$ dependencies$[\xi' \mapsto$ dependencies$(\xi') \cup \{\xi \subseteq \chi\}]$
- for all χ in X do analysis $:=$ analysis$[\chi \mapsto \chi]$
- analysis $:=$ analysis$[\beta \mapsto \{\,\}][\delta \mapsto \{\,\}]$
- (* iteration *)
- while worklist $\neq \{\,\}$ do
  - let $C_1 \uplus \{\xi \subseteq \chi\} =$ worklist
  - in do worklist $:= C_1$
    - if (analysis ξ) $\not\sqsubseteq$ analysis(χ) then do
      - analysis $:=$ analysis$[\chi \mapsto$ analysis$(\chi) \cup$ (analysis ξ)$]$
      - for all q in dependencies$[\chi]$ do
        - worklist $:=$ worklist $\cup \{q\}$
- (* finalisation *)
- return (analysis(β), analysis(δ))

Figure 12. Worklist algorithm for constraint solving.

After intialisation of the worklist set and the finite maps, the algorithm proceeds by considering constraints from the worklist as long as these are available. In each iteration a constraint is selected and tested for satisfaction. Here, we use the finite map analysis as a substitution and write analysis ξ for the interpretation of the flow term ξ under the subsitution provided by analysis. If a constraint is found unsatisfied, we update the solution for its right-hand-side flow variable χ and add all dependent constraints to the worklist. If the worklist is empty, the algorithm produces a pair consisting of the solutions for the flow variables β and δ. These are then guaranteed to consist of flow terms that, besides from applications and abstractions, are exclusively constructed from concrete labels and the flow variables from X.

7.3 Syntactic Correctness

A trivial observation about the completion algorithm from Figure 8 with respect to the defintions from Section 6 is the following:

Lemma 6. For all types τ, there is a fully parametric $\widehat{\tau}$, such that $\mathbf{C}(\tau, \varepsilon) = \widehat{\tau}$. ∎

Now, the correctness of both the reconstruction algorithm from Figure 7 and the worklist algorithm from Figure 12 with respect to the type and effect system from Section 5 comes in two parts. First, we have that each analysis produced by the algorithm is indeed admitted by the flow-typing rules of Figure 6.

Theorem 7 (Syntactic soundness). If we have that $\mathbf{R}(\widehat{\Gamma}, t) = (\widehat{\tau}, \beta, \delta, C)$ and $\mathbf{S}(C, \{\,\}, \beta, \delta) = (\psi, \varphi)$ for a fully flexible $\widehat{\Gamma}$, then $[\,] \mid \widehat{\Gamma} \vdash t : \widehat{\tau}^{\psi}\ \&\ \varphi$. ∎

Second, we have that the algorithm produces best analyses for all analysable terms. This result depends crucially on the invariant maintainind by the reconstruction algorithm, i.e., that R always produces fully flexible types. In particular, we have that the join algorithm from Figure 9 will not fail if it invoked with two fully flexible completions of a single underlying type.

Lemma 8. If $\widehat{\tau}_1$ and $\widehat{\tau}_2$ are fully flexible with $\lfloor \widehat{\tau}_1 \rfloor = \lfloor \widehat{\tau}_2 \rfloor = \tau$ for some τ, then $\mathbf{J}(\widehat{\tau}_1, \widehat{\tau}_2) = \widehat{\tau}$ with $\lfloor \widehat{\tau} \rfloor = \tau$. ∎

Similarly, the matching algorithm from Figure 11 is guaranteed to succeed when invoked with one fully flexible and one fully parametric completion of the same underlying type:

Lemma 9. If $\widehat{\tau}$ is fully flexible and $\widehat{\tau}'$ fully parametric with $\lfloor \widehat{\tau} \rfloor = \lfloor \widehat{\tau}' \rfloor$, then $\mathbf{M}([\,], \widehat{\tau}, \widehat{\tau}') = \theta$ with $\theta\, \widehat{\tau}' \equiv \widehat{\tau}$. ∎

Theorem 10 (Syntactic completeness). If $[\,] \mid \widehat{\Gamma} \vdash t : \widehat{\tau}' \,\&\, \varphi'$ with $\widehat{\Gamma}$ fully flexible, then there are $\widehat{\tau}$, β, δ, C, ψ, and φ with $\mathbf{R}(\widehat{\Gamma}, t) = (\widehat{\tau}, \beta, \delta, C)$ and $\mathbf{S}(C, \{\,\}, \beta, \delta) = (\psi, \varphi)$ and $(\widehat{\tau}, \psi, \varphi)$ a best analysis for t in $\widehat{\Gamma}$. ∎

8. Related Work

Early approaches to flow analysis for higher-order languages, e.g., the closure analysis of Sestoft (1991) and the set-based analysis of Heintze (1994), were monovariant, allowing only a single, context-insensitive analysis result to be associated with each of a program's functions. Later work resulted in polyvariant analyses that allow for the analysis results associated with at least some identifiers in a program to be context-sensitive; examples include Shivers' k-CFA (1991) and Nielson and Nielson's infinitary analysis (1997).

Polymorphic type and effect systems for flow analysis, such as Fähndrich's (2008), typically restrict polyvariant analysis results to be associated with let-bound identifiers only, leaving function parameters to be analysed monovariantly. Exceptions are the approaches of Faxén (1997) and Smith and Wang (2000), who also present polymorphic type and effect systems for flow analysis that allow for function parameters to be analysed polyvariantly rather than monovariantly. The most important difference between our approach and both the approach of Faxén and that of Smith and Wang is that, while we propose a single analysis, Faxén and Smith and Wang investigate families of constraint systems parameterised over inference strategies; the choices of strategies that lead to decidable analyses in their systems are rather ad hoc. Furthermore, the look-and-feel of the systems of Faxén and Smith and Wang differs significantly from ours, as we are, to the best of our knowledge, the first to consider the use of first-class operators on effects and annotations. Gustavsson and Svenningsson (2001) propose constrained type schemes that show a superficial similarity to ours, but do not allow quantification over effect operators; moreover, they do not allow type schemes to be associated with lambda-bound identifiers.

An important class of type-based flow analyses makes use of intersection types rather than polymorphic types. In general, intersection types allow for more fine-grained analysis results than polymorphic types (Wells et al. 2002). Kfoury and Wells (1999) show that inference is decidable if analyses are restricted to intersection types of finite rank. Their inference algorithm makes essential use of so-called expansion variables and is arguably much more complicated than the one we give for our analysis in Section 7. Banerjee and Jensen (2003) demonstrate that the restriction to rank-2 intersection types allows for a simpler algorithm, but only at the expense of decreased precision, while Mossin (2003) proceeds in the opposite direction and shows that exact flow analyses can be obtained at the expense of a nonelementary recursive inference problem.

A major advantage of the use of intersection types is that they admit *principal typings* rather than mere *principal types* (Jim 1996). As type systems with principal typings allow for terms to be typed independently from the types of their free variables, analyses based on intersection typing are even more modular than systems with just principal types. Our type and effect system does not admit principal typings, but, interestingly, in practice, the same level of modularity can be achieved as for systems with intersection types. That is, if, for a given term, the *underlying* types of its free variables are given, rather than their *annotated* types, an analysis can be computed for which the best analysis for that term in any given annotated type environment is a substitution instance. More precisely, if for a given term t, we are given an underlying type environment Γ, such that $\Gamma \vdash t : \tau$ for some type τ, then Σ, $\widehat{\Gamma}$, $\widehat{\tau}$, ψ, and φ can be computed, such that $\Sigma \mid \widehat{\Gamma} \vdash t : \widehat{\tau}^{\psi} \,\&\, \varphi$ with $\lfloor \widehat{\Gamma} \rfloor = \Gamma$ and $\lfloor \widehat{\tau} \rfloor = \tau$, and, moreover, for each fully flexible $\widehat{\Gamma}'$ with $\lfloor \widehat{\Gamma}' \rfloor = \Gamma$, there is a computable substitution θ mapping annotation variables to annotations and effect variables to effects, such that $(\theta\widehat{\tau}, \theta\psi, \theta\varphi)$ is a best analysis for t in $\widehat{\Gamma}'$. The idea is to first tentatively "guess" a fully parametric completion of the given underlying type environment and then, as flow inference proceeds, to gradually adapt this completion by "growing" a substitution on flow variables. Then, effectively, our type and effect system admits, in a sense, principal typings, but only as far as annotations and effects are concerned. For practical purposes, this suffices, because, as real-world higher-order functional languages are typically based on the Damas-Milner typing discipline, which itself does not admit principal typings, underlying type environments can be expected to be available for all terms under analysis.

The increased precision obtained from the use of polymorphic recursion in type-based analyses, as realised by Dussart et al. (1995), is reported on by several authors, including Henglein and Mossin (1994), and Tofte and Talpin (1994). To the best of our knowledge, we are the first to consider the generalisation to polymorphic types for all function arguments rather than for just those of functions from which fixed points are obtained.

9. Conclusions and Further Work

In this paper, we have presented a type and effect system for flow analysis with higher-ranked polymorphic types and higher-order effect operators. This system allows us to attain precision beyond what is offered by the ML-style let-polymorphic types that are typically used in polymorphic effect systems. The key innovation of our work is the use of fully flexible types, i.e., types that are as polymorphic as possible but impose no restrictions on the arguments that can be passed to functions. Given fully flexible types for all free variables, our analysis, which is a conservative extension of the standard Damas-Milner typing discipline, admits "best analyses" for all programs analysable: such analyses are both precise and modular.

Our analysis distinguishes between producers and consumers. In the present paper we have focused on producers and consumers for Boolean and function values, but our approach applies to other data types as well. In particular, although the details are syntactically rather heavy, our analysis can be extended to user-defined, algebraic data types, as found in modern functional languages such as Haskell and ML. Accounting for the use of let-polymorphism in the underlying type system is largely an orthogonal issue.

The flow analysis presented in this paper is a typical forward analysis: we keep track of the flow from producers to consumers. As future work—and as part of our research agenda to develop a reusable framework that can be used to construct precise and modular type and effect systems, much like monotone frameworks (Kam and Ullman 1977) are used to construct data-flow analyses—we aim at formulating a backward variation of our analysis, in which we keep track, for each production site, at which program points constructed values are consumed.

Many static analyses for higher-order languages can, in a type-based formulation, be expressed as variations on flow analysis. We expect our approach to be of value to these analyses as well and, hence, we plan to define higher-ranked polymorphic type and effect systems for analyses such as binding-time analysis, strictness

analysis, and usage analysis, and to compare the results obtained with those from existing let-polymorphic systems.

If a polyvariant type-based analysis is used to drive an optimising program transformation, a trade-off arises between the modularity of the analysis and the effectiveness of the transformation. For let-polymorphism, this trade-off may be resolved by differentiating between local and global let-bound identifiers (Holdermans and Hage 2010). For higher-ranked polymorphism, a similar measure may be in order, i.e., to obtain more effective transformations, selected lambda-bound identifiers may have to receive non-fully parametric types. Investigating how the algorithm of Section 7 can be adapted to such scenarios is a challenging but nevertheless appealing direction for further work.

Finally, characterising the difference in expressiveness and the trade-offs in implementation techniques between our analysis and systems based on intersection types of various ranks promises to be an interesting topic for further research.

Acknowledgments

This work was supported by the Netherlands Organisation for Scientific Research through its project on "Scriptable Compilers" (612.063.406) and carried out while the first author was employed at Utrecht University. The authors would like to thank Arie Middelkoop and Jeroen Weijers for their helpful comments on a draft of this paper, and the anonymous reviewers for their insightful feedback on the submitted version.

References

Anindya Banerjee and Thomas P. Jensen. Modular control-flow analysis with rank 2 intersection types. *Mathemathical Structures in Computer Science*, 13(1):87–124, 2003.

Luís Damas and Robin Milner. Principal type-schemes for functional programs. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, January 1982*, pages 207–212. ACM Press, 1982.

Dirk Dussart, Fritz Henglein, and Christian Mossin. Polymorphic recursion and subtype qualifications: Polymorphic binding-time analysis in polynominal time. In Alan Mycroft, editor, *Static Analysis, Second International Symposium, SAS'95, Glasgow, UK, September 27, 1995, Proceedings*, volume 983 of *Lecture Notes in Computer Science*, pages 118–135. Springer-Verlag, 1995.

Manuel Fähndrich and Jakob Rehof. Type-based flow analysis and context-free language reachability. *Mathematical Structures in Computer Science*, 18(5):823–894, 2008.

Karl-Filip Faxén. Polyvariance, polymorphism and flow analysis. In Mads Dam, editor, *Analysis and Verification of Multiple-Agent Languages, 5th LOMAPS Workshop, Stockholm, Sweden, June 24–26, 1996, Selected Papers*, volume 1192 of *Lecture Notes in Computer Science*, pages 260–278. Springer-Verlag, 1997.

Jörgen Gustavsson and Josef Svenningsson. Constraint abstractions. In Olivier Danvy and Andrzej Filinski, editors, *Programs as Data Objects, Second Symposium, PADO 2001, Aarhus, Denmark, May 21–23, 2001, Proceedings*, volume 2053 of *Lecture Notes in Computer Science*, pages 63–83. Springer-Verlag, 2001.

Nevin Heintze. Set-based analysis of ML programs. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming, Orlando, Florida, USA, 27–29 June 1994*, pages 306–317. ACM Press, 1994.

Fritz Henglein and Christian Mossin. Polymorphic binding-time analysis. In Donald Sannella, editor, *Programming Languages and Systems, ESOP'94, 5th European Symposium on Programming, Edinburgh, U.K., April 11–13, 1994, Proceedings*, volume 788 of *Lecture Notes in Computer Science*, pages 287–301. Springer-Verlag, 1994.

Stefan Holdermans and Jurriaan Hage. On the rôle of minimal typing derivations in type-driven program transformation, 2010. To appear in the proceedings of the 10th Workshop on Language Descriptions, Tools, and Applications (LDTA 2010), Paphos, Cyprus, 27–28 March 2010.

Trevor Jim. What are principal typings and what are they good for? In *Conference Record of POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Language, Papers Presented at the Symposium, St. Petersburg Beach, Florida, 21–24 January 1996*, pages 42–53. ACM Press, 1996.

John B. Kam and Jeffrey D. Ullman. Monotone data flow analysis frameworks. *Acta Informaticae*, 7:305–317, 1977.

Assaf J. Kfoury and Jerzy Tiuryn. Type reconstruction in finite rank fragments of the second-order λ-calculus. *Information and Computation*, 98(2):228–257, 1992.

Assaf J. Kfoury and Joe B. Wells. Principality and decidable type inference for finte-rank intersection types. In *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, January 20–22, 1999, San Antonio, TX*, pages 161–174. ACM Press, 1999.

Christian Mossin. Exact flow analysis. *Mathematical Structures in Computer Science*, 13(1):125–156, 2003.

Alan Mycroft. Polymorphic type schemes and recursive definitions. In Manfred Paul and Bernard Robinet, editors, *International Symposium on Programming, 6th Colloquium, Toulouse, April 17–19, 1984, Proceedings*, volume 167 of *Lecture Notes in Computer Science*, pages 217–228. Springer-Verlag, 1984.

Flemming Nielson and Hanne Riis Nielson. Type and effect systems. In Ernst-Rüdiger Olderog and Bernhard Steffen, editors, *Correct System Design, Recent Insight and Advances, (to Hans Langmaack on the occasion of his retirement from his professorship at the University of Kiel)*, volume 1710 of *Lecture Notes in Computer Science*, pages 114–136. Springer-Verlag, 1999.

Hanne Riis Nielson and Flemming Nielson. Infinitary control flow analysis: A collecting semantics for closure analysis. In *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15–17 January 1997*, pages 332–345. ACM Press, 1997.

Jens Palsberg. Type-based analysis and applications. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE'01, Snowbird, Utah, USA, June 18–19, 2001*, pages 20–27. ACM Press, 2001.

Peter Sestoft. *Analysis and Efficient Implementation of Functional Languages*. PhD thesis, University of Copenhagen, 1991.

Olin Shivers. *Control-flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, 1991.

Scott F. Smith and Tiejun Wang. Polyvariant flow analysis with constrained types. In Gert Smolka, editor, *Programming Languages and Systems, 9th European Symposium on Programming, ESOP 2000, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000,*

Berlin, Germany, March 25–April 2, 2000, Proceedings, volume 1782 of *Lecture Notes in Computer Sciences*, pages 382–396. Springer-Verlag, 2000.

Yan Mei Tang and Pierre Jouvelot. Effect systems with subtyping. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation, La Jolla, California, USA, June 21–23, 1995*, pages 45–53. ACM Press, 1995.

Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Conference Record of POPL'94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon, January 17–21, 1994*, pages 188–201. ACM Press, 1994.

Keith Wansbrough and Simon Peyton Jones. Once upon a polymorphic type. In *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, January 20–22, 1999, San Antonio, TX*, pages 15–28. ACM Press, 1999.

Joe B. Wells, Allyn Dimock, Robert Muller, and Franklyn A. Turbak. A calculus with polymorphic and polyvariant flow types. *Journal of Functional Programming*, 12(3):183–227, 2002.