Boolean Constraints for Binding-Time Analysis

Kevin Glynn, Peter J. Stuckey, Martin Sulzmann and Harald Søndergaard
{keving,pjs,sulzmann,harald}@cs.mu.oz.au

Department of Computer Science and Software Engineering,
The University of Melbourne, Victoria 3010, Australia

Abstract. To achieve acceptable accuracy, many program analyses for functional programs are "property polymorphic". That is, they can infer different input-output relations for a function at separate applications of the function, in a manner similar to type inference for a polymorphic language. We extend a property polymorphic (or "polyvariant") method for binding-time analysis, due to Dussart, Henglein, and Mossin, so that it applies to languages with ML-style type polymorphism. The extension is non-trivial and we have implemented it for Haskell. While we follow others in specifying the analysis as a non-standard type inference, we argue that it should be realised through a translation into the well-understood domain of Boolean constraints. The expressiveness offered by Boolean constraints opens the way for smooth extensions to sophisticated language features and it allows for more accurate analysis.

1 Introduction

The aim of this paper is to assimilate sophisticated program analysis capabilities in the context of a modern functional programming language with structured data and ML-style polymorphism. The most important capability of interest is "property polymorphism", that is, an analyser's ability to infer different properties for a definition f at separate uses of f, in a manner similar to type inference for a polymorphic language. For many program analyses, we need property polymorphism to achieve acceptable accuracy in the analysis. A second feature that we want is modularity of analysis, that is, the ability to produce analysis results that are context-independent, so as to support separate compilation.

"Property polymorphism" has previously been studied for a variety of program analyses for higher-order functional programs. However, usually, the underlying language misses features such as algebraic types and polymorphism. The assumption is usually that an extension, for example to ML-style polymorphism, is straightforward. Recent work, however, suggests that this is not necessarily so [15, 27]. One aim of this paper is to show that Boolean constraints give a better handle on such extensions.

In this work we take *binding-time* analysis as our example. This analysis is used for program specialisation. The purpose of binding-time analysis is to identify program expressions that can be evaluated at specialisation time, based on information about which parts of the program's input will be available at that time. Values that are known at specialisation-time are called *static*, while values that will only be known later are called *dynamic*.

Our interest in binding-time analysis is mainly as an example of a non-trivial program analysis in functional programming. More generally we are interested in program analysis expressed as inference of "constrained types". In Section 8 we discuss related analyses to which our methodology also applies.

In the context of binding-time analysis, "binding-time polymorphism" is often referred to as *polyvariance*. Our analysis is polyvariant and extends the analysis proposed by Dussart, Henglein and Mossin [7] to polymorphically typed programs. To investigate our method's practicality, we have implemented our binding-time analysis for the Glasgow Haskell Compiler (GHC). This assigns binding-time properties to all expressions, but we do not have a specialiser.

Much work on type-based analysis of functional programs introduces new constraint systems to define properties of interest, proceeding to explain how constraint solving or simplification is done for the new system. In many cases, to obtain accurate analysis, it appears necessary to employ non-standard versions of esoteric type systems such as intersection types.

We deviate from this path by deliberately seeking to utilise well-known constraint domains of great expressiveness. There are advantages of using well-known constraint domains:

- Useful known theorems can be used to simplify the presentation or improve the algorithms. For example, much is known about the complexity of constraint solving for various fragments of propositional logic.
- Extra expressiveness may assist extensions of the analysis to new language features. In Section 6 we argue that the extension to ML-style polymorphism is facilitated by the constraint view, and claim that this view helps us navigate the design space for the analysis.
- Extra expressiveness may allow for more accurate analysis. This is borne out by experience from other other programming language paradigms [2].
- An implementation may utilise efficient representations and algorithms. (For this paper, we have experimented with two different Boolean solvers.)

In the binding-time analysis literature we find many examples of monomorphic analysis for a polymorphic language, for example Mogensen's analysis [16], as well as polyvariant analysis for a monomorphic language [7]. The situation is similar for other analyses; for example, recent progress in usage analysis deals with either usage-monomorphic analysis for a polymorphic language [27] or usage-polymorphic analysis for a monomorphic language [28]. To our knowledge, this is the first time polyvariant binding-time analysis has been developed and implemented for a polymorphic language. Also, to the best of our knowledge, we present the first formalisation of the underlying binding-time logic. Owing to limited space, we concentrate on the core of our analysis and its novel aspects. For example, we leave out the treatment of structured data, focusing on basic types and function types.

The next section introduces basic concepts concerning types and constraints. Section 3 introduces the binding-time constraint system and logic. Section **??**

shows how to translate binding-time constraints to Boolean constraints. In Section 5 we extend the binding-time inference system of Dussart, Henglein and Mossin to polymorphically typed programs. The use of Boolean constraints provides us with a large design space. In Section 6 we describe alternative methods for supporting polymorphic function application. In Section 7 we discuss a method for finding fixed points by adding constraints. Section 8 discusses an implementation and concludes.

2 Preliminaries

2.1 The underlying type system

Programs are assumed to be well-typed in an *underlying* type system which we will refer to as UL. We consider an ML-style let-polymorphic typed language with base types **Int** and **Bool**:

$$
\begin{array}{ll}
\textbf{Types} & t ::= \alpha \mid \mathbf{Int} \mid \mathbf{Bool} \mid t \rightarrow t \\
\textbf{Type Schemes} & \sigma ::= t \mid \forall \bar{\alpha}.t \\
\textbf{Expressions} & e ::= x \mid \lambda x.e \mid e\, e \mid \mathsf{let}\ x = e\ \mathsf{in}\ e \mid x \sharp\ \bar{t}
\end{array}
$$

Expressions also include numerals and Boolean values True and False. We use vector notation for sequences. For example, $\bar{\alpha}$ represents a sequence $\alpha_1, \ldots \alpha_n$ of type variables. Types in UL are referred to as *underlying* types.

We assume well-typed expressions to be explicitly typed. An example well-formed, typed expression is

$$\mathsf{let}\ \mathsf{f} = \lambda x.x :: \forall \alpha.\alpha \rightarrow \alpha\ \mathsf{in}\ ((\mathsf{f} \sharp\ \mathbf{Int})\ 1 :: \mathbf{Int},\ (\mathsf{f} \sharp\ \mathbf{Bool})\ \mathsf{True} :: \mathbf{Bool})$$

We find it helpful to have polymorphic application (denoted by $\sharp$) explicit in the language, but we let polymorphic abstraction be implicit (restricted to let-statements), to avoid cluttering expressions.

2.2 Binding-time types

In line with Dussart, Henglein and Mossin, we impose a "binding-time type" structure on top of the underlying type structure. For instance, $S \overset{S}{\mapsto} D$ describes a static function that takes a static value of base type as an argument and returns a dynamic value of base type. The structure of binding-time types reflects the structure of the underlying type system.

$$
\begin{array}{ll}
\textbf{Annotations} & b ::= \delta \mid S \mid D \\
\textbf{Binding-Time Types} & \tau ::= \beta \mid b \mid \tau \overset{b}{\mapsto} \tau \\
\textbf{Binding-Time Type Schemes} & \eta ::= \tau \mid \forall \bar{\beta}, \bar{\delta}.C \Rightarrow \tau
\end{array}
$$

Note that we distinguish between annotation variables δ (which may only be instantiated to S or D) and binding-time type variables β (which may be instantiated to any τ, including δ). In practice it is not necessary for an implementation to distinguish the two, but for clarity we will do so in this presentation. We write $\bar{\beta}, \bar{\delta} = fv(\eta)$ to refer to the free binding-time and annotation variables in η. We now describe the constraint component C in binding-time type schemes η.

$$(\text{Sta})\ C \vdash (S \leq_a b) \qquad (\text{Dyn})\ C \vdash (b \leq_a D)$$

$$(\text{Hyp})\ C_1, (b_1 \leq_a b_2), C_2 \vdash (b_1 \leq_a b_2)$$

$$(\text{Refl})\ C \vdash (b \leq_a b) \qquad (\text{Trans})\ \frac{C \vdash (b_1 \leq_a b_2) \quad C \vdash (b_2 \leq_a b_3)}{C \vdash (b_1 \leq_a b_3)}$$

$$(\text{Bas}_w)\ C \vdash \mathit{wft}(b) \qquad (\text{Arrow}_w)\ \frac{\begin{array}{c} C \vdash (b_3 \leq_f \tau_1) \quad C \vdash \mathit{wft}(\tau_1) \\ C \vdash (b_3 \leq_f \tau_2) \quad C \vdash \mathit{wft}(\tau_2) \end{array}}{C \vdash \mathit{wft}(\tau_1 \overset{b_3}{\mapsto} \tau_2)}$$

$$(\text{Bas}_f)\ \frac{C \vdash (b_1 \leq_a b_2)}{C \vdash (b_1 \leq_f b_2)} \qquad (\text{Arrow}_f)\ \frac{C \vdash (b_1 \leq_a b_2)}{C \vdash (b_1 \leq_f \tau_1 \overset{b_2}{\mapsto} \tau_3)}$$

$$(\text{Bas}_s)\ \frac{C \vdash (b_1 \leq_a b_2)}{C \vdash (b_1 \leq_s b_2)} \qquad (\text{Arrow}_s)\ \frac{\begin{array}{c} C \vdash (b_2 \leq_a b_5) \\ C \vdash (\tau_4 \leq_s \tau_1) \quad C \vdash (\tau_3 \leq_s \tau_6) \end{array}}{C \vdash (\tau_1 \overset{b_2}{\mapsto} \tau_3 \leq_s \tau_4 \overset{b_5}{\mapsto} \tau_6)}$$

Fig. 1. Binding-Time Constraint Rules

2.3 Binding-time constraints

Relationships among binding-time variables are captured through constraints. Primitive constraints are of the form $(x \leq y)$, read as "y is at least as dynamic as x". There are various sorts: an ordering on annotations $(\cdot \leq_a \cdot)$, a structural ordering $(\cdot \leq_s \cdot)$ on binding-time types, and an auxiliary ordering $(\cdot \leq_f \cdot)$ described below.

In addition, constraints can be formed using conjunction and existential quantification. More precisely, the constraint language is given by the grammar

$$C ::= (\tau \leq \tau) \mid \mathit{wft}(\tau) \mid C \wedge C \mid \exists \bar{\beta}, \bar{\delta}.C$$

For binding-time types it is necessary to add an additional kind of constraint, that binding-time types are "well-formed". If the top-most annotation of a binding-time type is *dynamic* then we know nothing of its components and so they must all be *dynamic* too. For instance, $S \overset{D}{\mapsto} S$ is meaningless. The rules for constraints $\mathit{wft}(\cdot)$ and $(\cdot \leq_f \cdot)$ ensure that binding-time types are well-formed. Relation $(b_1 \leq_f \tau)$ ensures that b_1 is smaller than the top-most annotation of τ. For monomorphic underlying types, all constraints are reducible to $(\cdot \leq_a \cdot)$ constraints.

Figure 1 defines the primitive fragment of the binding-time constraint system, **BTC**. This is identical to the rules presented by Dussart, Henglein and

Mossin. We differ, however, by explicitly allowing for existential quantification over binding-time type variables in constraints. The rules governing the existential quantification are taken from cylindric constraint systems [21]. That is,

1. $C \vdash \exists\bar{\beta}, \bar{\delta}.C$
2. if $C_1 \vdash C_2$ then $\exists\bar{\beta}, \bar{\delta}.C_1 \vdash \exists\bar{\beta}, \bar{\delta}.C_2$
3. $\exists\bar{\beta}, \bar{\delta}.(C_1 \wedge \exists\bar{\beta}, \bar{\delta}.C_2) \equiv \exists\bar{\beta}, \bar{\delta}.C_1 \wedge \exists\bar{\beta}, \bar{\delta}.C_2$
4. $\exists\bar{\beta}_1, \bar{\delta}_1.\exists\bar{\beta}_2, \bar{\delta}_2.C \equiv \exists\bar{\beta}_2, \bar{\delta}_2.\exists\bar{\beta}_1, \bar{\delta}_1.C$

The purpose of the cylindrification operation $\exists\bar{\beta}, \bar{\delta}$ is to hide (or discard) the variables $\bar{\beta}, \bar{\delta}$. During inference of binding-time types we may introduce intermediate variables, not free in the resulting binding-time types or environments. At certain stages during inference, constraints that have been imposed on these variables will no longer be of interest, and the cylindrification operator can then be employed to simplify constraints. Inference will in fact be using Boolean constraints. When we later translate binding-time constraints to a Boolean form, the existential quantification will turn into existential quantification over Boolean variables.

Example 1. The identity function $\lambda x.x$ has the polymorphic type $\forall \alpha.\alpha \to \alpha$. Its binding-time type is $\forall \beta_1, \beta_2, \delta.(\beta_1 \leq \beta_2) \wedge (\delta \leq \beta_1) \wedge (\delta \leq \beta_2) \Rightarrow \beta_1 \overset{\delta}{\mapsto} \beta_2$ where the constraint $(\beta_1 \leq \beta_2) \wedge (\delta \leq \beta_1) \wedge (\delta \leq \beta_2)$ describes the binding-time behaviour of the identity function. The first conjunct says that output from the function is dynamic if input is. The other two conjuncts express a well-formedness condition. Note that the binding-time type variables β_1 and β_2 must range over the same type structure as variable α. An instance of $\lambda x.x$'s type is $(\mathbf{Int} \to \mathbf{Int}) \to (\mathbf{Int} \to \mathbf{Int})$ with

$$(D \overset{S}{\mapsto} S) \overset{S}{\mapsto} (D \overset{S}{\mapsto} S)$$

a possible binding-time type. This expresses that $\lambda x.x$ can take a function of binding-time type $D \overset{S}{\mapsto} S$ and the application's context can treat the result as being of the same binding-time type. Another possible binding-time type is

$$(D \overset{S}{\mapsto} S) \overset{S}{\mapsto} (D \overset{D}{\mapsto} D)$$

showing how the binding-time type system allows for coercion: It is acceptable for a static value to be treated as dynamic, if required by the context.

2.4 Shapes

To relate an expression's valid binding-time type to its underlying type, we employ a "shape" system. In Example 1, the identity function's binding-time type $\forall \beta_1, \beta_2, \delta.C \Rightarrow \beta_1 \overset{\delta}{\mapsto} \beta_2$ has shape $\forall \alpha.\alpha \to \alpha$. Formally, a shape environment Δ maps a polymorphic binding-time type variable to its corresponding underlying polymorphic type variable. If $\Delta = \{\beta_{11} : \alpha_1, \ldots, \beta_{nk_n} : \alpha_n\}$ then $domain(\Delta) = \bar{\beta}$

$$(\mathbf{Bool}_\delta)\ \Delta \vdash \delta : \mathbf{Bool} \qquad (\mathbf{Int}_\delta)\ \Delta \vdash \delta : \mathbf{Int}$$

$$(\mathbf{Bool}_S)\ \Delta \vdash S : \mathbf{Bool} \qquad (\mathbf{Bool}_D)\ \Delta \vdash D : \mathbf{Bool}$$

$$(\mathbf{Int}_S)\ \Delta \vdash S : \mathbf{Int} \qquad (\mathbf{Int}_D)\ \Delta \vdash D : \mathbf{Int}$$

$$(\beta)\ \frac{(\beta : \alpha) \in \Delta}{\Delta \vdash \beta : \alpha} \qquad (\text{Arrow})\ \frac{\Delta_1 \vdash \tau_1 : t_1 \quad \Delta_2 \vdash \tau_2 : t_2}{\Delta_1 \cup \Delta_2 \vdash \tau_1 \overset{b}{\mapsto} \tau_2 : t_1 \to t_2}$$

$$(\forall)\ \frac{\begin{array}{l} \Delta' = \{\beta_{11} : \alpha_1, \ldots, \beta_{1k_1} : \alpha_1 \\ \quad \vdots \\ \quad \beta_{n1} : \alpha_n, \ldots, \beta_{nk_n} : \alpha_n\} \end{array} \qquad \Delta \cup \Delta' \vdash \tau : t \quad \bar{\alpha} \notin range(\Delta)}{\Delta \vdash \forall \bar{\beta}, \bar{\delta}.C \Rightarrow \tau : \forall \bar{\alpha}.t}$$

Fig. 2. Shape rules

is the *domain* of Δ, and $range(\Delta) = \bar{\alpha}$ is the *range*. The judgement $\Delta \vdash \eta : \sigma$ states that under shape environment Δ the binding-time type η has shape σ. A judgement $\Delta \vdash \eta : \sigma$ is valid if it can be derived by the shape rules in Figure 2.

For simplicity, we usually omit information such as $\{\beta_1 : \alpha, \beta_2 : \alpha\}$ in a type scheme like $\forall \beta_1, \beta_2, \delta.(\beta_1 \leq \beta_2) \wedge (\delta \leq \beta_1) \wedge (\delta \leq \beta_2) \Rightarrow \beta_1 \overset{\delta}{\mapsto} \beta_2$. Shape information is easily recovered by inspection of an expression's underlying type and the corresponding binding-time type.

Shape inference will be important when performing binding-time analysis. Given an underlying type t, we will sometimes want the most general shape environment Δ and binding-time type τ such that $\Delta \vdash \tau : t$. We say (Δ, τ) is *more general* than (Δ', τ') iff $\Delta \vdash \tau : t$, $\Delta' \vdash \tau' : t$ and there exists a substitution ϕ such that $\phi\Delta = \Delta'$ and $\phi\tau = \tau'$.

Lemma 1. *Given a type t, one can compute the most general shape environment Δ and binding-time type τ such that $\Delta \vdash \tau : t$.*

Lemma 1 ensures that there exists an algorithm which computes the most general shape environment Δ and binding-time type τ given an underlying type t. In our presentation, we treat the algorithm as a deduction system with clauses of the form $t \vdash (\tau, \Delta)$ where t is the input value and Δ and τ are the output values. Similarly, there is an algorithm which computes the most general binding-time type τ, given a shape environment Δ and an underlying type t (we write $\Delta, t \vdash \tau$). Finally, there is an algorithm to compute the most general shape environment Δ given a binding-time type τ and an underlying type t (we write $\tau, t \vdash \Delta$).

3 Binding-Time Logic

We assume expressions to be well-typed in the underlying type system. Binding-time properties of expressions are specified by typing judgements which are of the form $C, \Gamma \vdash (e :: \sigma) : \eta$ where C is in **BTC**, Γ is a binding-time type environment assigning binding-time types to the free variables of e, $(e :: \sigma)$ is a type-annotated expression, and η is e's binding-time type. Always implicit, is a shape environment Δ associating binding-time type variables with underlying type variables.

We assume that we can translate a program's initial type environment (which maps primitive functions to their underlying types), into an initial binding-time environment mapping primitive functions to their binding-time types. We require that the binding-time type of a primitive function is "directed":

Definition 1 (Well-shapedness, Polarities, Directedness). Given a type scheme $\sigma = \forall \bar{\alpha}.t$, a binding-time type $\eta = \forall \bar{\beta}.C \Rightarrow \tau$, a shape environment Δ, a constraint C', and a binding-time type τ'.

- We say that η is *well-shaped* (wrt. σ and Δ) iff $\Delta \vdash \eta : \sigma$.
- We say that (C', η) is well-shaped iff η is well-shaped.
- We define *polarities* recursively: β appears in *positive* position in τ' (written $\tau'[\beta^+]$) iff $\tau' = \beta$, or else $\tau' = \tau_1 \stackrel{\delta}{\mapsto} \tau_2$ and either $\tau_1[\beta^-]$ or $\tau_2[\beta^+]$. Similarly β appears in *negative* position in τ' (written $\tau'[\beta^-]$) iff $\tau' = \tau_1 \stackrel{\delta}{\mapsto} \tau_2$ and $\tau_1[\beta^+]$ or $\tau_2[\beta^-]$.
- We say that (C', η) is *directed* (wrt. σ and Δ) iff (1) η is well-shaped, and (2) there exists an extension $\Delta' \supseteq \Delta$ such that $\Delta' \vdash \tau : t$ and (3) for each β_1, β_2 such that $C \wedge C' \vdash (\beta_1 \leq_s \beta_2)$, (a) $\Delta' \vdash \beta_1 : \alpha$ and $\Delta' \vdash \beta_2 : \alpha$ (that is, β_1 and β_2 have same type according to Δ') and (b) $\tau[\beta_1^-]$, and $\tau[\beta_2^+]$ (that is, β_1 occurs in negative position and β_2 occurs in positive position).
- We say that η is directed iff (true, η) is directed.

These definitions extend to type environments in the natural way.

The directedness condition simply states that all constraints of the form $(\beta_1 \leq_s \beta_2)$ describe relations between input values β_1 and output values β_2. This is naturally fulfilled by a binding-time analysis, since intuitively, information flows from negative to positive positions.

Example 2. A directed binding-time description of if–then–else ($\mathsf{ite} :: \mathbf{Bool} \to \alpha \to \alpha \to \alpha$) is as follows:

$$\mathsf{ite} : \forall \delta, \beta_1, \beta_2, \beta_3.(\beta_1 \leq_s \beta_3) \wedge (\beta_2 \leq_s \beta_3) \wedge (\delta \leq_f \beta_3) \Rightarrow \delta \stackrel{S}{\mapsto} \beta_1 \stackrel{S}{\mapsto} \beta_2 \stackrel{S}{\mapsto} \beta_3$$

Note that the constraint $(\delta \leq_f \beta_3)$ vacuously satisfies the directedness condition.

The constraints C that can appear in type schemes and on the left hand side of the turnstile are restricted to those generated by the grammar:

$$C ::= (b_1 \leq_a b_2) \mid (\tau_1 \leq_s \tau_2) \mid (b \leq_f \tau) \mid \mathit{wft}(\tau) \mid C \wedge C \mid \exists \bar{\beta}, \bar{\delta}.C$$

$$\text{(Var)} \quad \frac{(x:\eta) \in \Gamma \quad \Delta \vdash \eta : \sigma}{C, \Gamma \vdash (x :: \sigma) : \eta}$$

$$\text{(Sub)} \quad \frac{C, \Gamma \vdash (e :: t) : \tau_2 \quad C \vdash (\tau_2 \leq_s \tau_1) \quad C \vdash \mathit{wft}(\tau_1)}{C, \Gamma \vdash (e :: t) : \tau_1}$$

$$\text{(Abs)} \quad \frac{C, \Gamma_x.x : \tau_1 \vdash (e :: t_2) : \tau_2 \quad C \vdash \mathit{wft}(\tau_1) \quad \Delta \vdash \tau_1 : t_1}{C, \Gamma_x \vdash (\lambda x.e :: t_1 \to t_2) : \tau_1 \overset{S}{\mapsto} \tau_2}$$

$$\text{(App)} \quad \frac{C, \Gamma \vdash (e_1 :: t_1 \to t_2) : (\tau_1 \overset{b}{\mapsto} \tau_2) \quad C, \Gamma \vdash (e_2 :: t_1) : \tau_1}{C, \Gamma \vdash (e_1\ e_2 :: t_2) : \tau_2}$$

$$\text{(Let)} \quad \frac{C, \Gamma_x \vdash (e_1 :: \sigma) : \eta \quad C, \Gamma_x.x : \eta \vdash (e_2 :: t) : \tau}{C, \Gamma_x \vdash (\mathsf{let}\ x = (e_1 :: \sigma)\ \mathsf{in}\ e_2 :: t) : \tau}$$

$$(\exists\text{I}) \quad \frac{C, \Gamma \vdash (e :: t) : \tau \quad \bar{\beta}, \bar{\delta} = \mathit{fv}(C) \backslash \mathit{fv}(\Gamma, \tau)}{\exists \bar{\beta}, \bar{\delta}.C, \Gamma \vdash (e :: t) : \tau}$$

$$(\forall\text{I}) \quad \frac{\begin{array}{c} C \wedge D, \Gamma \vdash (e :: t) : \tau \quad \Delta \vdash \tau : t \quad \bar{\beta}, \bar{\delta} \subseteq \mathit{fv}(D, \tau) \backslash \mathit{fv}(\Gamma, C) \\ \text{where} \quad \Delta \vdash \beta_{ij} : \alpha_i \end{array}}{C \wedge \exists \bar{\beta}, \bar{\delta}.D, \Gamma \vdash (e :: \forall \bar{\alpha}.t) : \forall \bar{\beta}, \bar{\delta}.D \Rightarrow \tau}$$

$$(\forall\text{E}) \quad \frac{\begin{array}{c} C, \Gamma \vdash (x :: \forall \bar{\alpha}.t) : \forall \bar{\beta}, \bar{\delta}.D \Rightarrow \tau \\ \Delta \vdash \tau : t \quad \mathit{inst}(\Delta, \bar{t}, \bar{\alpha}) = \bar{\tau} \\ C \vdash [\bar{\tau}/\bar{\beta}, \bar{b}/\bar{\delta}]D \end{array}}{C, \Gamma \vdash ((x \sharp \bar{t}) :: [\bar{t}/\bar{\alpha}]t) : [\bar{\tau}/\bar{\beta}, \bar{b}/\bar{\delta}]\tau}$$

$$\text{(Fix)} \quad \frac{C, \Gamma_x.x : \eta \vdash (e :: t) : \eta \quad C \vdash \mathit{wft}(\eta) \quad \Delta \vdash \eta : t}{C, \Gamma_x \vdash ((\mathsf{fix}\ x :: t\ \mathsf{in}\ e) :: t) : \eta}$$

Fig. 3. Binding-Time Typing Rules

where τ_1 and τ_2 must have the same shape. This excludes, for example, constraints of the form $(\beta \leq_s S \overset{S}{\mapsto} D)$. Such a restriction is necessary, as we do not allow for subtyping in the underlying type system UL.

Figure 3 defines the binding-time logic. Γ_x denotes the environment $\{y : \eta \in \Gamma \mid y \neq x\}$. Most of the rules are straightforward. For example, rule (Sub) expresses the obvious subsumption (or coercion) rule for binding-time types. Rule (Abs) says that, in order to deduce the binding-time type $\tau_1 \overset{S}{\mapsto} \tau_2$ for an expression $(\lambda x.e :: t_1 \to t_2)$, we must be able to deduce τ_2 for $(e :: t_2)$ under the additional assumption that x has type τ_1 (which is well-formed and of shape t_1). Rule ($\exists$I) allows unnecessary binding-time type variables in the constraint component of a typing judgement to be discarded. In rule ($\forall$I), we require that quantification over β variables is restricted to βs related to polymorphic variables α by the shape environment.

Rule ($\forall$E) allows instantiation of binding-time type schemes, at a variable's usage site. We need to compute the corresponding binding-time type instances for each underlying type instance. Let Δ be a shape environment, $\bar{t}$ a sequence of underlying types, and $\bar{\alpha}$ a sequence of underlying type variables. We define $inst(\Delta, \bar{t}, \bar{\alpha}) = \bar{\tau}$ where $\bar{\tau}$ are fresh binding-time types of appropriate shape: Each element of $\bar{\tau}$ is related to the corresponding element of $\bar{\beta}$ by the shape environment Δ. That is, $\Delta, t_i \vdash \tau_{ij}$ where $\Delta \vdash \beta_{ij} : \alpha_i$.

A judgement $C, \Gamma \vdash (e :: \sigma) : \eta$ is *valid* if the judgement is derivable by the rules in Figure 3.

Lemma 2 (Conservative Extension). *Let e be well-typed with type σ and let the binding-time environment Γ be well-formed and directed. Then $C, \Gamma \vdash (e :: \sigma) : \eta$ for some constraint C and binding-time type η.*

From here on we consider only expressions that are well-typed in UL.

The following lemma allows us to ignore well-formedness constraints in examples. We will also assume from now on that all binding-time types are directed.

Lemma 3 (Well-Formed and Directed). *Let e be well-typed with type σ. If $C, \Gamma \vdash (e :: \sigma) : \eta$ and Γ is well-formed and directed, then (C, η) is well-formed and directed.*

We define an ordering on binding-time type schemes for use in the inference algorithm (read $C'' \vdash \eta \leq \eta'$ as η is more general than η' in the context of constraint C''):

$$C'' \vdash (\forall \bar{\beta}, \bar{\delta}. C \Rightarrow \tau) \leq (\forall \bar{\beta}', \bar{\delta}'. C' \Rightarrow \tau') \text{ iff } C'' \wedge C' \vdash \exists \bar{\beta}, \bar{\delta}. C \wedge (\tau \leq_s \tau').$$

(We assume here, with no loss of generality, that there are no name clashes.)

4 Translation to Boolean Constraints

Ultimately, binding-time constraints are nothing more than Boolean constraints. We can read the binding-time values S and D as *false* and *true*, respectively. δ variables are Boolean variables, and a constraint $(\delta_1 \leq_a \delta_2)$ is simply an implication $\delta_1 \rightarrow \delta_2$. Since all valid binding-time type constraints $(\tau_1 \leq_s \tau_2)$ have the same shape, every binding-time constraint can also be understood as a Boolean constraint. We map a binding-time constraint C to its corresponding Boolean constraint $[\![C]\!]$ as follows:

$$
\begin{aligned}
[\![S]\!] &= \mathit{false} \\
[\![D]\!] &= \mathit{true} \\
[\![\delta]\!] &= \delta \\
[\![(b_1 \leq_a b_2)]\!] &= [\![b_1]\!] \rightarrow [\![b_2]\!] \\
[\![(\delta \leq_f \beta)]\!] &= \delta \rightarrow \beta \\
[\![(b_1 \leq_f \tau_1 \overset{b_3}{\mapsto} \tau_2)]\!] &= [\![b_1]\!] \rightarrow [\![b_2]\!] \\
[\![(\tau_1 \overset{b_1}{\mapsto} \tau_1' \leq_s \tau_2 \overset{b_2}{\mapsto} \tau_2')]\!] &= [\![b_1]\!] \rightarrow [\![b_2]\!] \wedge [\![(\tau_1' \leq_s \tau_2')]\!] \wedge [\![(\tau_2 \leq_s \tau_1)]\!] \\
[\![(\beta \leq_s \beta')]\!] &= \beta \rightarrow \beta' \\
[\![\mathit{wft}(\tau_1 \overset{b_3}{\mapsto} \tau_2)]\!] &= [\![(b_3 \leq_f \tau_1)]\!] \wedge [\![\mathit{wft}(\tau_1)]\!] \ \wedge [\![(b_3 \leq_f \tau_2)]\!] \wedge [\![\mathit{wft}(\tau_2)]\!] \\
[\![C_1 \wedge C_2]\!] &= [\![C_1]\!] \wedge [\![C_2]\!] \\
[\![\exists \bar{\beta}, \bar{\delta}.C]\!] &= \exists \bar{\beta}, \bar{\delta}.[\![C]\!]
\end{aligned}
$$

We note that $C \vdash C'$ iff $[\![C]\!] \models [\![C']\!]$, where $\models$ denotes entailment in propositional logic.

Since the class of Boolean constraints generated is a subclass of the set HORN of propositional Horn clauses, an immediate advantage of the translation is that we have linear time algorithms for satisfiability [6], and hence many other operations. If more sophisticated analysis calls for a larger class of Boolean constraints then there are efficient representations and algorithms for Boolean constraints available, for example based on ROBDDs [3]. Finally, useful operations such as existential and universal quantification, conjunction and disjunction have a clear and well-understood meaning.

5 Binding-Time Inference

We assume that we are given a well-typed program, each subexpression annotated with its underlying type. Binding-time inference computes the missing binding-time information. The inference algorithm, Figure 4, is formulated as a deduction system over clauses of the form

$$\Gamma, e :: t \vdash_{\mathit{inf}} (C, \tau)$$

with a binding-time type environment Γ and a type-annotated expression e as input, and a binding-time constraint C and a binding-time type τ as output. An algorithm in style of algorithm $\mathcal{W}$ can be derived from this given specification in a straightforward way.

All rules are syntax-directed except rule ($\exists$ Intro). This rule is justified by the corresponding rule in the logical system. We assume that rule ($\exists$ Intro) is applied aggressively, so that useless variables do not appear in analysis results. Rule (Var-λ) handles lambda-bound variables whereas rule (Var–Inst) handles instantiation of let-bound variables. Considering the binding-time logic of Figure 3, the straightforward approach to polymorphism is to instantiate polymorphic binding-time variables in both constraints and binding-time types with a τ of the appropriate shape.

$$
\text{(Var–Inst)}\quad \frac{\begin{array}{c} x : \forall \bar{\beta}, \bar{\delta}.C \Rightarrow \tau \in \Gamma \quad t' = [\bar{t}/\bar{\alpha}]t \quad \tau, t \vdash \Delta \quad inst(\Delta, \bar{t}, \bar{\alpha}) = \bar{\tau} \quad \bar{\delta}' \text{ new} \\ C' = [\![[\bar{\tau}/\bar{\beta}, \bar{\delta}'/\bar{\delta}]C]\!] \quad \tau' = [\bar{\tau}/\bar{\beta}, \bar{\delta}'/\bar{\delta}]\tau \end{array}}{\Gamma, ((x :: \forall \bar{\alpha}.t)\sharp \bar{t}) :: t' \vdash_{inf} (C', \tau')}
$$

$$
\text{(Var–}\lambda\text{)}\quad \frac{(x : \tau) \in \Gamma \quad C = [\![(\tau \leq_s \tau') \wedge wft(\tau')]\!] \quad t \vdash (\Delta, \tau')}{\Gamma, x :: t \vdash_{inf} (C, \tau')}
$$

$$
\text{(Abs)}\quad \frac{\Gamma_x.x : \tau_1, e :: t_2 \vdash_{inf} (C, \tau_2) \quad t_1 \vdash (\Delta_1, \tau_1) \quad \delta \text{ new}}{\Gamma_x, \lambda x.e :: t_1 \to t_2 \vdash_{inf} (C \wedge [\![wft(\tau_1 \overset{\delta}{\mapsto} \tau_2)]\!]), \tau_1 \overset{\delta}{\mapsto} \tau_2)}
$$

$$
\text{(App)}\quad \frac{\begin{array}{c} \Gamma, e_1 :: t_1 \vdash_{inf} (C_1, \tau_1) \quad \Gamma, e_2 :: t_2 \vdash_{inf} (C_2, \tau_2) \\ C = C_1 \wedge C_2 \wedge [\![(\tau_1 \leq_s \tau_2 \overset{\delta}{\mapsto} \tau_3) \wedge wft(\tau_3)]\!] \\ \delta \text{ new} \quad t_3 \vdash (\Delta_3, \tau_3) \end{array}}{\Gamma, ((e_1 :: t_1)(e_2 :: t_2)) :: t_3 \vdash_{inf} (C, \tau_3)}
$$

$$
\text{(Let)}\quad \frac{\begin{array}{c} \Gamma_x, e_1 :: t_1 \vdash_{inf} (C_1, \tau_1) \\ gen(C_1, \Gamma_x, \tau_1, t_1, \bar{\alpha}) = (C_o, \eta_o) \quad \Gamma_x.x : \eta_o, e_2 :: t_2 \vdash_{inf} (C_2, \tau_2) \\ C = C_o \wedge C_2 \end{array}}{\Gamma_x, (\mathsf{let}\ x = (e_1 :: \forall \bar{\alpha}.t_1)\ \mathsf{in}\ e_2) :: t_2 \vdash_{inf} (C, \tau_2)}
$$

$$
\text{(Fix)}\quad \frac{\begin{array}{c} t \vdash (\Delta, \tau) \quad \bar{\delta} = fv(\tau) \quad \eta_0 = \forall \bar{\delta}.[\![wft(\tau)]\!] \Rightarrow \tau \\ \mathcal{F}(\Gamma_x.x : \eta_0, e :: t) = (C, \tau) \end{array}}{\Gamma_x, (\mathsf{fix}\ x :: t\ \mathsf{in}\ e) :: t \vdash_{inf} (C, \tau)}
$$

$$
(\exists\ \text{Intro})\quad \frac{\Gamma, e :: t \vdash_{inf} (C, \tau) \quad \bar{\beta}, \bar{\delta} = fv(C) \backslash fv(\Gamma, \tau)}{\Gamma, e :: t \vdash_{inf} (\exists \bar{\beta}, \bar{\delta}.C, \tau)}
$$

Fig. 4. Binding-Time Inference

Example 3. Consider the (type annotated) polymorphic application

$$((\mathsf{id} \sharp \mathbf{Int} \to \mathbf{Int}) :: (\mathbf{Int} \to \mathbf{Int}) \to (\mathbf{Int} \to \mathbf{Int}))\ (\mathsf{g} :: \mathbf{Int} \to \mathbf{Int}) :: \mathbf{Int} \to \mathbf{Int}$$

where $\mathsf{id} :: \forall \alpha.\alpha \to \alpha$. Here are binding-time descriptions for id, g and $(\mathsf{id}\ \mathsf{g})$:

$$
\begin{aligned}
\mathsf{id} &: ((\beta_1 \leq_s \beta_3) \wedge (\delta_2 \leq_f \beta_1) \wedge (\delta_2 \leq_f \beta_3), \beta_1 \overset{\delta_2}{\mapsto} \beta_3) \\
\mathsf{g} &: ((\delta_5 \leq_a \delta_4) \wedge (\delta_5 \leq_a \delta_6), \delta_4 \overset{\delta_5}{\mapsto} \delta_6) \\
\mathsf{id}\ \mathsf{g} &: \delta_7 \overset{\delta_8}{\mapsto} \delta_9
\end{aligned}
$$

We need to build an instance of id where all polymorphic binding-time variables of shape α are instantiated to binding-time types of shape ($\mathbf{Int} \to \mathbf{Int}$). We substitute $\delta_{10} \overset{\delta_{11}}{\mapsto} \delta_{12}$ for β_1, $\delta_{14} \overset{\delta_{15}}{\mapsto} \delta_{16}$ for β_3 and δ_{13} for δ_2, resulting in

$$(\mathsf{id} \sharp\ \mathbf{Int} \to \mathbf{Int})::(\mathbf{Int} \to \mathbf{Int}) \to (\mathbf{Int} \to \mathbf{Int}) : (\delta_{10} \overset{\delta_{11}}{\mapsto} \delta_{12}) \overset{\delta_{13}}{\mapsto} \delta_{14} \overset{\delta_{15}}{\mapsto} \delta_{16}$$

under the "instantiated" constraints

$$((\delta_{10} \overset{\delta_{11}}{\mapsto} \delta_{12}) \leq_s (\delta_{14} \overset{\delta_{15}}{\mapsto} \delta_{16})) \wedge (\delta_{13} \leq_f (\delta_{10} \overset{\delta_{11}}{\mapsto} \delta_{12})) \wedge (\delta_{13} \leq_f (\delta_{14} \overset{\delta_{15}}{\mapsto} \delta_{16}))$$

These constraints are translated into Boolean constraints using the constraint rules of Figure 1 and Section **??**. We then proceed to analyse (id g) by applying the usual inference rules.

Rule (Let) introduces annotation polymorphism as well as type polymorphism. Note that we can not simply generalise over all the free binding-time type variables because the expression might not be annotated with its principal type.

Example 4. Consider the following type-annotated program:

$\mathsf{g} :: \forall \alpha_1.\alpha_1 \to ((\alpha_1, \mathbf{Int}), (\alpha_1, \mathbf{Bool}))$
$\mathsf{g\ x} = \mathsf{let\ f} :: \forall \alpha_2.\alpha_1 \to \alpha_2 \to (\alpha_1, \alpha_2)$
$\qquad\qquad \mathsf{f\ y\ z} = \mathsf{(y,\ z)}$
$\qquad \mathsf{in\ ((f} \sharp\ \mathbf{Int}\mathsf{)\ x\ 1,\ (f} \sharp\ \mathbf{Bool}\mathsf{)\ x\ True)}$

Whatever binding-time variable β we assign to y, we are not allowed to quantify over β.

Therefore a polymorphic binding-time variable is only considered free if its corresponding polymorphic type variable is free.

We define a generalisation function giving the generalised type scheme and the generalised constraint. Let C be a constraint, Γ a type environment, τ a binding-time type, t an underlying type and $\bar{\alpha}$ a sequence of underlying type variables. Then

$$gen(C, \Gamma, \tau, t, \bar{\alpha}) = (\exists \bar{\beta}, \bar{\delta}.C,\ \forall \bar{\beta}, \bar{\delta}.C \Rightarrow \tau)$$

where

- $\tau, t \vdash \Delta$
- $\bar{\beta} = \{\beta \mid \Delta \vdash \beta : \alpha \text{ for some } \alpha \in \bar{\alpha}\}$
- $\bar{\delta} = fv(C, \tau) \backslash (fv(\Gamma) \cup domain(\Delta))$

We first associate all free binding-time type variables with all free underlying type variables. Then, a binding-time variable β is free if the corresponding type variable α is free. It remains to compute all free annotation variables $\bar{\delta}$. We further note that we push the whole constraint C into the type scheme (one could be more efficient by pushing in only the *affected* constraints).

In rules (Var-λ), (Abs), (App), and (Fix), given an underlying type t, we generate a most general binding-time type τ (the most general shape environment Δ is not of interest) such that $\Delta \vdash t : \tau$. Moreover, we add the constraint $wft(\tau)$ to the output constraint to ensure well-formedness of the newly generated binding-time type. For example, a λ-bound variable of type $\mathbf{Int} \rightarrow \mathbf{Int}$, could give rise to a binding-time type of $\delta_1 \overset{\delta_2}{\mapsto} \delta_3$. The constraint $\mathit{wft}(\delta_1 \overset{\delta_2}{\mapsto} \delta_3)$ reduces to $\delta_2 \rightarrow \delta_1 \wedge \delta_2 \rightarrow \delta_3$.

With (Fix), we follow Dussart, Henglein and Mossin [7], performing Kleene-Mycroft iteration until a fixed point is found. Define[1]

$$
\begin{array}{ll}
\mathcal{F}(\Gamma_x.x : \eta_i, e :: t) & \\
= \mathcal{F}(\Gamma_x.x : \eta_{i+1}, e :: t) & \text{if } \mathsf{true} \vdash \eta_i < \eta_{i+1} \\
= (C, \tau) & \text{if } \mathsf{true} \vdash \eta_i = \eta_{i+1}
\end{array}
$$

where $\Gamma_x.x : \eta_i, e :: t \vdash_{\mathit{inf}} (C, \tau)$ and $(\cdot, \eta_{i+1}) = \mathit{gen}(C, \Gamma_x.x : \eta_i, \tau)$.

Note that the sequence $\eta_o \leq \ldots \leq \eta_i \leq \ldots$ is necessarily finite: The binding-time constraint rules (Figure 1) ensure that only binding-time types of the same shape are comparable. Moreover, as the generated binding-time type schemes are of the form $\forall\bar{\delta}.C \Rightarrow \tau$, that is, the quantifier is over *annotation* variables only (not binding-time type variables), each only has a finite number of instances.

Example 5. Consider the binding-time analysis of the following program

$\mathsf{g} :: \mathbf{Bool} \rightarrow \mathbf{Int} \rightarrow \mathbf{Int} \rightarrow (\mathbf{Int}, \mathbf{Int})$
$\mathsf{g\ p\ x\ y = if\ p\ then\ (x,y)\ else\ (snd\ (g\ p\ y\ x),\ y)}$

or, in our syntax,

$$\mathsf{fix\ g\ in\ } \lambda\mathsf{p}.\lambda\mathsf{x}.\lambda\mathsf{y}.\mathsf{ite\ p\ (x, y)\ (snd\ (g\ p\ y\ x), y)}$$

where

- $(\mathsf{e}_1, \mathsf{e}_2)$ builds a pair from two expressions.
- snd returns the second element of a pair.
- ite is a built-in *if–then–else* function which returns a pair of **Int**s. (Two extensions of the algorithm, supported by our implementation, allow a more natural definition: (1) If–then–else can be written using a built-in *case* expression, and (2) polymorphic application allows us to use the usual if–then–else function and instantiate it to the pair in this context.)

Fixed point calculation starts with a fresh binding-time type scheme η_0 for g:

$$\mathsf{g} : \forall\bar{\delta}.C \Rightarrow \delta_1 \overset{\delta_7}{\mapsto} \delta_2 \overset{\delta_8}{\mapsto} \delta_3 \overset{\delta_9}{\mapsto} (\delta_4, \delta_5)^{\delta_6}$$

C consists entirely of well-formedness constraints:

$$\delta_7 \rightarrow \delta_1 \wedge \delta_7 \rightarrow \delta_8 \wedge \delta_8 \rightarrow \delta_2 \wedge \delta_8 \rightarrow \delta_9 \wedge \delta_9 \rightarrow \delta_3 \wedge \delta_9 \rightarrow \delta_6 \wedge \delta_6 \rightarrow \delta_4 \wedge \delta_6 \rightarrow \delta_5$$

[1] In terms of Boolean constraints, the binding-time type scheme ordering given at the end of Section 3 translates to: $(\forall\bar{\delta}_1.C_1 \Rightarrow \tau_1) \leq (\forall\bar{\delta}_2.C_2 \Rightarrow \tau_2)$ iff $C_2 \vdash \exists\bar{\delta}_1.(C_1 \wedge (\tau_1 \leq_s \tau_2))$. (Without loss of generality we assume there are no name clashes between $\bar{\delta}_1$ and $\bar{\delta}_2$.)

However, this constraint plays no further part in this example. Indeed, well-formedness constraints have no impact in this example, so to simplify the presentation, let us treat η_0 as simply

$$\forall\bar{\delta}.\mathsf{true} \Rightarrow \delta_1 \mapsto \delta_2 \mapsto \delta_3 \mapsto (\delta_4, \delta_5)^{\delta_6}$$

and consider that the initial binding-time environment Γ_0 contains:

$$\begin{array}{ll} \mathsf{snd}: \forall\bar{\delta}.\ \delta_2 \to \delta_4 \ \Rightarrow \ (\delta_1, \delta_2)^{\delta_3} \mapsto \delta_4 & \\ \mathsf{ite}: \ \forall\bar{\delta}.\delta_1 \to \delta_{10} \wedge \delta_2 \to \delta_8 \wedge \delta_3 \to \delta_9 \wedge \delta_4 \to \delta_{10} \wedge \delta_5 \to \delta_8 \wedge \delta_6 \to \delta_9 \wedge \delta_7 \to \delta_{10} & \\ \quad \Rightarrow \delta_1 \mapsto (\delta_2, \delta_3)^{\delta_4} \mapsto (\delta_5, \delta_6)^{\delta_7} \mapsto (\delta_8, \delta_9)^{\delta_{10}} & \end{array}$$

Similarly we shall ignore the variable renaming in the (Var-λ) rule.

Inference now proceeds as follows. The body of the lambda abstraction is analysed in the binding-time environment

$$\Gamma_1 \ = \ \Gamma_0, \mathsf{g}: \eta_0, \mathsf{p}: \delta_7, \mathsf{x}: \delta_8, \mathsf{y}: \delta_9$$

For the then-branch we have

$$\Gamma_1, (\mathsf{x}, \mathsf{y}) \vdash_{inf} (\mathsf{true}, (\delta_8, \delta_9)^{\delta_{10}})$$

That is, no constraints are contributed by this subexpression. For the else-branch, consider first the sub-expression $\mathsf{g\ p\ y\ x}$. Three applications of rule (App) effectively introduce a binding-time type $(\delta_{11}, \delta_{12})^{\delta_{13}}$ for the sub-expression, together with the constraint

$$(\delta_{14} \mapsto \delta_{15} \mapsto \delta_{16} \mapsto (\delta_{17}, \delta_{18})^{\delta_{19}} \leq_s \delta_7 \mapsto \delta_9 \mapsto \delta_8 \mapsto (\delta_{11}, \delta_{12})^{\delta_{13}})$$

Notice how a fresh binding-time type $\delta_{14} \mapsto \delta_{15} \mapsto \delta_{16} \mapsto (\delta_{17}, \delta_{18})^{\delta_{19}}$ has been introduced via rule (Var-Inst).

The constraint translates to

$$\delta_7 \to \delta_{14} \wedge \delta_9 \to \delta_{15} \wedge \delta_8 \to \delta_{16} \wedge \delta_{17} \to \delta_{11} \wedge \delta_{18} \to \delta_{12} \wedge \delta_{19} \to \delta_{13}$$

However, since variables δ_{14} to δ_{19} are of no further interest (they are neither free in the environment, nor in the result type $(\delta_{11}, \delta_{12})^{\delta_{13}}$), we can existentially quantify (rule ($\exists$ Intro)):

$$\exists\delta_{14} \cdots \delta_{19}.\ \delta_7 \to \delta_{14} \wedge \delta_9 \to \delta_{15} \wedge \delta_8 \to \delta_{16} \wedge \delta_{17} \to \delta_{11} \wedge \delta_{18} \to \delta_{12} \wedge \delta_{19} \to \delta_{13}$$

which is equivalent to true. Hence we have

$$\Gamma_1, \mathsf{g\ p\ y\ x} \vdash_{inf} (\mathsf{true}, (\delta_{11}, \delta_{12})^{\delta_{13}})$$

Similarly, for $\mathsf{snd}\ (\mathsf{g\ p\ y\ x})$ we introduce the result type δ_{20}, and generate the constraint

$$\delta_{22} \to \delta_{24} \wedge [\![(\delta_{21}, \delta_{22})^{\delta_{23}} \mapsto \delta_{24} \leq_s (\delta_{11}, \delta_{12})^{\delta_{13}} \mapsto \delta_{20})]\!]$$

Again, once we have translated to a Boolean constraint and eliminated uninteresting variables, we are left with the vacuous constraint true:

$$\Gamma_1, \mathsf{snd}(\mathsf{g\ p\ y\ x}) \vdash_{inf} (\mathsf{true}, \delta_{20})$$

It follows that

$$\Gamma_1, (\mathsf{snd}(\mathsf{g\ p\ y\ x}), \mathsf{y}) \vdash_{inf} (\mathsf{true}, (\delta_{20}, \delta_9)^{\delta_{25}})$$

Calling the result of g's body $(\delta_{26}, \delta_{27})^{\delta_{28}}$, and introducing a fresh instance of the type for ite, we get

$$\begin{aligned}\llbracket(\delta_{29} \mapsto (\delta_{30}, \delta_{31})^{\delta_{32}} \mapsto (\delta_{33}, \delta_{34})^{\delta_{35}} \mapsto (\delta_{36}, \delta_{37})^{\delta_{38}} \\ \leq_s \delta_7 \mapsto (\delta_8, \delta_9)^{\delta_{10}} \mapsto (\delta_{20}, \delta_9)^{\delta_{25}} \mapsto (\delta_{26}, \delta_{27})^{\delta_{28}})\rrbracket\end{aligned}$$

with the additional constraint

$$\delta_{29} \to \delta_{38} \wedge \delta_{30} \to \delta_{36} \wedge \delta_{31} \to \delta_{37} \wedge \delta_{32} \to \delta_{38} \wedge \delta_{33} \to \delta_{36} \wedge \delta_{34} \to \delta_{37} \wedge \delta_{35} \to \delta_{38}$$

The structural constraint translates to

$$\begin{array}{l}\delta_7 \to \delta_{29} \wedge \delta_8 \to \delta_{30} \wedge \delta_9 \to \delta_{31} \wedge \delta_{10} \to \delta_{32} \wedge \delta_{20} \to \delta_{33} \wedge \\ \delta_9 \to \delta_{34} \wedge \delta_{25} \to \delta_{35} \wedge \delta_{36} \to \delta_{26} \wedge \delta_{37} \to \delta_{27} \wedge \delta_{38} \to \delta_{28}\end{array}$$

However, only $\delta_7, \delta_8, \delta_9, \delta_{26}, \delta_{27}$, and δ_{28} are of interest. Existential quantification over the remaining variables yields

$$\delta_7 \to \delta_{28} \wedge \delta_8 \to \delta_{26} \wedge \delta_9 \to \delta_{27}$$

For the lambda abstraction we therefore obtain

$$\begin{array}{l}\Gamma_0, \mathsf{g} : \eta_0, \lambda\mathsf{p}.\lambda\mathsf{x}.\lambda\mathsf{y}.\mathsf{ite\ p\ (x,y)\ (snd\ (g\ p\ y\ x), y)} \\ \qquad \vdash_{inf} \quad (\delta_7 \to \delta_{28} \wedge \delta_8 \to \delta_{26} \wedge \delta_9 \to \delta_{27}, \delta_7 \mapsto \delta_8 \mapsto \delta_9 \mapsto (\delta_{26}, \delta_{27})^{\delta_{28}})\end{array}$$

Generalising, we have a new binding-time type scheme η_1 for g:

$$\mathsf{g} : \forall\bar{\delta}.\delta_7 \to \delta_{28} \wedge \delta_8 \to \delta_{26} \wedge \delta_9 \to \delta_{27} \Rightarrow \delta_7 \mapsto \delta_8 \mapsto \delta_9 \mapsto (\delta_{26}, \delta_{27})^{\delta_{28}}$$

(ignoring well-formedness constraints).

Since $\eta_1 \not\leq \eta_0$, the fixed point calculation needs to continue. The effect of η_1 is to add new constraints when analysing $\mathsf{g\ p\ y\ x}$. We leave it as an exercise for the reader to show that η_1 is a fixed point. (Hint: snd removes the constraints we did not already have.)

We can state soundness and completeness results for well-typed programs. Soundness means that for every deduction in the inference system, we can find an "equivalent" deduction in the logic. We note that the constraints derived in the inference system are Boolean constraints whereas constraints in the logical system are in **BTC**. Therefore, equivalence here means that the constraints derived in both deduction systems are (semantically) equivalent after translating the constraints in the logical system into Boolean constraints.

Theorem 1 (Soundness of Inference). *Let $\Gamma, e :: \sigma \vdash_{inf} (C, \tau)$. Then $C', \Gamma \vdash (e :: \sigma) : \eta$ for some C' such that $gen(C, \Gamma, \tau) = (C_o, \eta)$ and $[\![C']\!] = C_o$.*

Completeness states that every deduction derivable in the logical system is subsumed by a deduction in the inference system.

Theorem 2 (Completeness of Inference). *Let $C, \Gamma \vdash (e :: \sigma) : \forall \bar{\beta}_1, \bar{\delta}_1.C_1 \Rightarrow \tau_1$. Let $\Gamma, e :: \sigma \vdash_{inf} (C_2, \tau_2)$. Then $[\![C \wedge C_1]\!] \models \exists \bar{\beta}_2, \bar{\delta}_2.(C_2 \wedge [\![(\tau_2 \leq_s \tau_1)]\!])$ where $\bar{\beta}_2, \bar{\delta}_2 = fv(C_2, \tau_2) \backslash fv(\Gamma)$.*

In addition, we can state that inference yields principal types.

Definition 2. Given a pair $(\Gamma, e :: \sigma)$ consisting of a binding-time environment Γ and a type-annotated expression $(e :: \sigma)$, together with a pair (C, η), where C is a binding-time constraint and η is a binding-time type. (C, η) is a *principal type* of $(\Gamma, e :: \sigma)$ iff

1. $C, \Gamma \vdash (e :: \sigma) : \eta$
2. whenever $C', \Gamma \vdash (e :: \sigma) : \eta'$, we have $C' \vdash C$ and $C' \vdash \eta \leq \eta'$.

(The ordering on binding-time type schemes was defined at the end of Section 3.) Note that principality is defined with respect to a given type-annotated expression. Hence, annotating an expression with a more general (underlying) type may result in a more general binding-time type. A principal binding-time type represents a class of types which are semantically equivalent. This is in contrast to [4] where principal types are syntactically unique.

Corollary 1 (Principal Types). *Let $\eta = \forall \bar{\beta}, \bar{\delta}.C \Rightarrow \tau$. Assume*

$$\mathsf{true}, \emptyset \vdash (e :: \sigma) : \eta$$

where e is a closed expression. Let (true, η) be the principal type of $(\emptyset, e :: \sigma)$. Then $\emptyset, e :: \sigma \vdash_{inf} (C, \tau)$.

6 Alternative Methods for Handling Polymorphism

With certain constraint solvers the instantiate method (rule (Var-Inst)) is impractical. They store the constraints in an optimised internal form; to reverse engineer this representation to find which relations hold between polymorphic variables may be very inefficient, if not impossible.

However, it is always possible for the solver to determine if a constraint set entails a particular relationship amongst variables. Many solvers can do this efficiently. The Test-Add method uses entailment to avoid constraint instantiation.

6.1 The Test-Add method

Instead of instantiating variables in the constraint component of the type scheme $\forall\bar{\beta},\bar{\delta}.C \Rightarrow \tau$, the "Test–Add" method queries which implications hold between elements of $\bar{\beta}$ in C. For those that hold we add inequalities between the corresponding instantiating τs to C.

Note we have two forms of constraints involving polymorphic binding-time variables. $\beta \to \beta'$ (connects two polymorphic binding-time variables), and $\delta \to \beta$ (a well-formedness condition or generated from a conditional expression such as if-then-else).

Consider a binding-time type scheme $\eta = \forall\bar{\beta},\bar{\delta}.C \Rightarrow \tau$, an underlying type scheme $\sigma = \forall\bar{\alpha}.t$, a shape environment Δ and a sequence of instantiation types τ_{ij} such that $\Delta \vdash \tau : t$ and for each $\Delta \vdash \beta_{ij} : \alpha_i$ it holds that $\Delta \vdash \tau_{ij} : \alpha_i$. We define

$$\begin{aligned}
\mathcal{T}_{\alpha_i}(\Delta,\tau,\bar{\tau}_i) &= \left\{ [\![(\tau_{ij} \leq_s \tau_{ik})]\!] \;\middle|\; \begin{array}{l} \Delta \vdash \beta_{ij} : \alpha_i, \tau[\beta_{ij}^{-}], \\ \Delta \vdash \beta_{ik} : \alpha_i, \tau[\beta_{ik}^{+}], \\ C \models (\beta_{ij} \to \beta_{ik}) \end{array} \right\} \\
\mathcal{T}(\Delta,\tau,\bar{\tau}) &= \textstyle\bigwedge_{\alpha_i \in range(\Delta)} \mathcal{T}_{\alpha_i}(\Delta,\tau,\bar{\tau}_i)
\end{aligned}$$

Recall that we only consider directed constraints (Definition 1) and the inference rules preserve this condition.

The set of generated constraints is a subclass of HORN. Hence, the test $C \models (\beta_{ij} \to \beta_{ik})$ can be performed in linear time. It remains to handle constraints of the form $(\delta \leq_f \beta)$. In Example 3 we had the constraint $(\delta_2 \leq_f \beta_1)$ where δ_2 was instantiated to δ_{13} and β_1 to $\delta_{10} \overset{\delta_{11}}{\mapsto} \delta_{12}$. We define

$$\begin{aligned}
\mathcal{A}_{\alpha_i}(\bar{\beta}_i,\bar{\tau}_i) &= \textstyle\bigwedge\{\beta_{ij} \to [\![|\tau_{ij}|]\!]\} \\
\mathcal{A}(\Delta,\bar{\beta},\bar{\tau}) &= \textstyle\bigwedge_{\alpha_i \in range(\Delta)} \mathcal{A}_{\alpha_i}(\bar{\beta}_i,\bar{\tau}_i)
\end{aligned}$$

where $|\tau|$ refers to τ's top-level annotation and is defined as follows:

$$|b| = b \qquad |\beta| = \beta \qquad |\tau \overset{b}{\mapsto} \tau'| = b$$

This ensures that if we have a relation $\delta \to \beta_{ij}$ in the constraint component of the type scheme, then together with $\mathcal{A}_{\alpha_i}$ we obtain $\delta \to |\tau_{ij}|$.

Example 6. We reconsider Example 3. Since $\beta_1 \to \beta_3$ holds in id's binding-time type, the $\mathcal{T}$ operator yields the constraint $[\![(\delta_{10} \overset{\delta_{11}}{\mapsto} \delta_{12} \leq_s \delta_{14} \overset{\delta_{15}}{\mapsto} \delta_{16})]\!]$. The $\mathcal{A}$ operator yields the constraint $\beta_1 \to \delta_{11} \wedge \beta_3 \to \delta_{15}$. Adding id's constraint component $[\![(\beta_1 \leq_s \beta_3) \wedge (\delta_2 \leq_f \beta_1) \wedge (\delta_2 \leq_f \beta_3)]\!]$, we obtain exactly the same result as the instantiation method.

The following states that "Instantiate" and "Test–Add" are equivalent.

Lemma 4. *Given a binding-time type scheme $\eta = \forall \bar{\beta}, \bar{\delta}.C \Rightarrow \tau$, an underlying type scheme $\sigma = \forall \bar{\alpha}.t$, a shape environment Δ and a sequence of instantiation types τ_{ij} such that $\Delta \vdash \tau : t$ and for each $\Delta \vdash \beta_{ij} : \alpha_i$ it holds that $\Delta \vdash \tau_{ij} : \alpha_i$.*

$$[\![[\bar{\tau}/\bar{\beta}]C]\!] \equiv \exists \bar{\beta}.([\![C]\!] \wedge \mathcal{T}(\Delta, \tau, \bar{\tau}) \wedge \mathcal{A}(\Delta, \bar{\beta}, \bar{\tau})).$$

Lemma 4 allows us to replace rule (Var–Inst) with the equivalent rule (Var–Inst′):

$$(\text{Var–Inst}')\ \frac{\begin{array}{c} x : \forall \bar{\beta}, \bar{\delta}.C \Rightarrow \tau \in \Gamma \quad t' = [\bar{t}/\bar{\alpha}]t \\ \tau, t \vdash \Delta \quad inst(\Delta, \bar{t}, \bar{\alpha}) = \bar{\tau} \\ C' = [\![[\bar{\delta}'/\bar{\delta}]C]\!] \wedge \mathcal{T}(\Delta, \tau, \bar{\tau}) \wedge \mathcal{A}(\Delta, \bar{\beta}, \bar{\tau}) \\ \tau' = [\bar{\tau}/\bar{\beta}, \bar{\delta}'/\bar{\delta}]\tau \quad \bar{\delta}' \text{ new} \end{array}}{\Gamma, ((x :: \forall \bar{\alpha}.t) \sharp\ \bar{t}) :: t' \vdash_{inf} (C', \tau')}$$

Theorem 3 (Polymorphic Application 1). *Let Γ be directed. Assume $\Gamma, e :: t \vdash_{inf} (C, \tau)$ using rule (Var–Inst) while $\Gamma, e :: t \vdash_{inf} (C', \tau')$ using rule (Var–Inst′). Then $C \equiv \pi(C')$ and $\tau = \pi(\tau')$ for some renaming π.*

6.2 The Match method

We briefly comment on further alternative methods. For more details we refer to [10]. In "Test–Add" we query possible relations between binding-time variables which are related by the shape environment to the same underlying type variable. We may wish to avoid this querying during the actual analysis. The "Match" method allows us to simply add constraints which will give a similar effect.

The method works by matching the type component of the variable's type scheme with the requested type at the instantiation site.

Example 7. Consider again Example 3. First generate a new binding-time type of the desired form: $(\delta_{10} \overset{\delta_{11}}{\mapsto} \delta_{12}) \overset{\delta_{13}}{\mapsto} \delta_{14} \overset{\delta_{15}}{\mapsto} \delta_{16}$. Instead of substituting the appropriate types for β_1, β_3 and δ_2, generate constraints as follows:

$$\begin{array}{l} (\beta_1 \overset{\delta_2}{\mapsto} \beta_3 \leq_s (\delta_{10} \overset{\delta_{11}}{\mapsto} \delta_{12}) \overset{\delta_{13}}{\mapsto} (\delta_{14} \overset{\delta_{15}}{\mapsto} \delta_{16})) \\ \equiv (\delta_{10} \overset{\delta_{11}}{\mapsto} \delta_{12} \leq_s \beta_1) \wedge (\beta_3 \leq_s \delta_{14} \overset{\delta_{15}}{\mapsto} \delta_{16}) \wedge \delta_2 \rightarrow \delta_{13} \end{array}$$

Note that the polymorphic binding-time variables β_1 and β_3 result from the polymorphic variable α. How do we translate the structural constraints into Boolean constraints? Since in this case id's constraints support $\beta_1 \rightarrow \beta_3$ we should have that $(\delta_{10} \overset{\delta_{11}}{\mapsto} \delta_{12} \leq_s \delta_{14} \overset{\delta_{15}}{\mapsto} \delta_{16})$. If id did not support $\beta_1 \rightarrow \beta_3$ then we should not have this constraint. This reasoning is captured by the constraint:

$$\begin{array}{l} (\beta_1 \rightarrow \beta_3) \rightarrow [\![((\delta_{10} \overset{\delta_{11}}{\mapsto} \delta_{12} \leq_s \delta_{14} \overset{\delta_{15}}{\mapsto} \delta_{16}))]\!] \\ \equiv (\beta_1 \rightarrow \beta_3) \rightarrow (\delta_{14} \rightarrow \delta_{10} \wedge \delta_{11} \rightarrow \delta_{15} \wedge \delta_{12} \rightarrow \delta_{16}) \end{array}$$

7 Constraint Based Fixed Points

If we are willing to give up some accuracy in the presence of polymorphic recursion in binding-time types then we can replace the (Fix) rule by a simpler rule which just ensures that the constraints generated will be above the least fixed point, while avoiding a fixed point computation.

The following rule forces all recursive invocations to have the same binding-time type. This must be a super-type of the binding-time type of the recursively defined expression.

$$\text{(FixC)}\ \frac{\begin{array}{c}\Delta, t \vdash \tau' \quad \Gamma_x.x : \tau', e :: t \vdash_{inf} (C_1, \tau) \quad \Delta, t \vdash \tau'' \\ C = C_1 \wedge [\![(\tau \leq_s \tau')]\!] \wedge [\![(\tau \leq_s \tau'')]\!] \wedge [\![\mathit{wft}(\tau')]\!] \wedge [\![\mathit{wft}(\tau'')]\!]\end{array}}{\Gamma_x, (\mathsf{fix}\ x :: t\ \mathsf{in}\ e) :: t \vdash_{inf} (C, \tau'')}$$

The "shortcut" of stipulating $(\tau \leq_s \tau')$ may have the unfortunate side-effect of introducing constraints amongst the arguments to a function. An example of this phenomenon appears in Example 8 below. A simple way of eliminating such constraints is to couch the result in terms of a fresh binding-time type τ'', with the constraint $(\tau \leq_s \tau'')$.

The following shows how this approach may lose accuracy, finding a correct but inaccurate binding-time type for a function.

Example 8. Consider again g from Example 5:

$$\begin{array}{l}\mathsf{g} :: \mathbf{Bool} \to \mathbf{Int} \to \mathbf{Int} \to (\mathbf{Int}, \mathbf{Int}) \\ \mathsf{g\ p\ x\ y = if\ p\ then\ (x,y)\ else\ (snd\ (g\ p\ y\ x),\ y)}\end{array}$$

The binding-time type inferred using the rule (Fix) was

$$\mathsf{g} : \forall\bar{\delta}.\delta_7 \to \delta_{28} \wedge \delta_8 \to \delta_{26} \wedge \delta_9 \to \delta_{27} \Rightarrow \delta_7 \mapsto \delta_8 \mapsto \delta_9 \mapsto (\delta_{26}, \delta_{27})^{\delta_{28}}$$

This time, the initial environment Γ has

$$\mathsf{g} : \delta_1 \mapsto \delta_2 \mapsto \delta_3 \mapsto (\delta_4, \delta_5)^{\delta_6}$$

so that $\delta_1 \ldots \delta_6$ are not generic. For the subexpression g p y x, we now get

$$(\delta_1 \mapsto \delta_2 \mapsto \delta_3 \mapsto (\delta_4, \delta_5)^{\delta_6} \leq_s \delta_7 \mapsto \delta_9 \mapsto \delta_8 \mapsto (\delta_{11}, \delta_{12})^{\delta_{13}})$$

This translates to

$$\delta_7 \to \delta_1 \wedge \delta_9 \to \delta_2 \wedge \delta_8 \to \delta_3 \wedge \delta_4 \to \delta_{11} \wedge \delta_5 \to \delta_{12} \wedge \delta_6 \to \delta_{13}$$

and none of the variables involved can be discarded. For snd (g p y x) we introduce the result type δ_{14}, and add the constraint

$$\delta_{16} \to \delta_{18} \wedge [\![(\delta_{15}, \delta_{16})^{\delta_{17}} \mapsto \delta_{18} \leq_s (\delta_{11}, \delta_{12})^{\delta_{13}} \mapsto \delta_{14})]\!]$$

that is,

$$\delta_{16} \to \delta_{18} \wedge \delta_{11} \to \delta_{15} \wedge \delta_{12} \to \delta_{16} \wedge \delta_{13} \to \delta_{17} \wedge \delta_{18} \to \delta_{14}$$

Here variables $\delta_{11} \ldots \delta_{13}$ and $\delta_{15} \ldots \delta_{18}$ can be discarded. Existentially quantifying over these leaves us with

$$\delta_7 \to \delta_1 \wedge \delta_9 \to \delta_2 \wedge \delta_8 \to \delta_3 \wedge \delta_5 \to \delta_{14}$$

So

$$\Gamma, \mathsf{snd}(\mathsf{g\ p\ y\ x}) \vdash_{inf} (\delta_7 \to \delta_1 \wedge \delta_9 \to \delta_2 \wedge \delta_8 \to \delta_3 \wedge \delta_5 \to \delta_{14},\ \delta_{14})$$

It follows that

$$\Gamma, (\mathsf{snd}(\mathsf{g\ p\ y\ x}), \mathsf{y}) \vdash_{inf} (\delta_7 \to \delta_1 \wedge \delta_9 \to \delta_2 \wedge \delta_8 \to \delta_3 \wedge \delta_5 \to \delta_{14},\ (\delta_{14}, \delta_9)^{\delta_{19}})$$

Calling the result of g's body $(\delta_{20}, \delta_{21})^{\delta_{22}}$, we get for the ite expression

$$\begin{aligned} [\![(\delta_{23} \mapsto (\delta_{24}, \delta_{25})^{\delta_{26}} \mapsto (\delta_{27}, \delta_{28})^{\delta_{29}} \mapsto (\delta_{30}, \delta_{31})^{\delta_{32}} \\ \leq_s \delta_7 \mapsto (\delta_8, \delta_9)^{\delta_{10}} \mapsto (\delta_{14}, \delta_9)^{\delta_{19}} \mapsto (\delta_{20}, \delta_{21})^{\delta_{22}})]\!] \end{aligned}$$

with the additional constraint

$$\delta_{23} \to \delta_{32} \wedge \delta_{24} \to \delta_{30} \wedge \delta_{25} \to \delta_{31} \wedge \delta_{26} \to \delta_{32} \wedge \delta_{27} \to \delta_{30} \wedge \delta_{28} \to \delta_{31} \wedge \delta_{29} \to \delta_{32}$$

The structural constraint translates to

$$\begin{aligned} &\delta_7 \to \delta_{23} \wedge \delta_8 \to \delta_{24} \wedge \delta_9 \to \delta_{25} \wedge \delta_{10} \to \delta_{26} \wedge \delta_{14} \to \delta_{27} \wedge \\ &\delta_9 \to \delta_{28} \wedge \delta_{19} \to \delta_{29} \wedge \delta_{30} \to \delta_{20} \wedge \delta_{31} \to \delta_{21} \wedge \delta_{32} \to \delta_{22} \end{aligned}$$

Variables of interest are $\delta_1 \ldots \delta_9$ and $\delta_{20} \ldots \delta_{22}$. Eliminating the rest, we get

$$\delta_7 \to \delta_1 \wedge \delta_9 \to \delta_2 \wedge \delta_8 \to \delta_3 \wedge \delta_5 \to \delta_{20} \wedge \delta_7 \to \delta_{22} \wedge \delta_8 \to \delta_{20} \wedge \delta_9 \to \delta_{21}$$

For the lambda abstraction we therefore obtain the result

$$\begin{aligned} (\delta_7 \to \delta_1 \wedge \delta_9 \to \delta_2 \wedge \delta_8 \to \delta_3 \wedge \delta_5 \to \delta_{20} \wedge \delta_7 \to \delta_{22} \wedge \delta_8 \to \delta_{20} \wedge \delta_9 \to \delta_{21}, \\ \delta_7 \mapsto \delta_8 \mapsto \delta_9 \mapsto (\delta_{20}, \delta_{21})^{\delta_{22}}) \end{aligned}$$

At this point, the rule (FixC) adds the following two inequalities

$$[\![(\delta_7 \mapsto \delta_8 \mapsto \delta_9 \mapsto (\delta_{20}, \delta_{21})^{\delta_{22}} \leq_s \delta_1 \mapsto \delta_2 \mapsto \delta_3 \mapsto (\delta_4, \delta_5)^{\delta_6})]\!]$$

and

$$[\![(\delta_7 \mapsto \delta_8 \mapsto \delta_9 \mapsto (\delta_{20}, \delta_{21})^{\delta_{22}} \leq_s \delta_{33} \mapsto \delta_{34} \mapsto \delta_{35} \mapsto (\delta_{36}, \delta_{37})^{\delta_{38}})]\!]$$

Altogether we then have

$$\begin{aligned} &\delta_7 \to \delta_1 \wedge \delta_9 \to \delta_2 \wedge \delta_8 \to \delta_3 \wedge \delta_5 \to \delta_{20} \wedge \delta_7 \to \delta_{22} \wedge \delta_8 \to \delta_{20} \wedge \delta_9 \to \delta_{21} \wedge \\ &\delta_1 \to \delta_7 \wedge \delta_2 \to \delta_8 \wedge \delta_3 \to \delta_9 \wedge \delta_{20} \to \delta_4 \wedge \delta_{21} \to \delta_5 \wedge \delta_{22} \to \delta_6 \wedge \\ &\delta_{33} \to \delta_7 \wedge \delta_{34} \to \delta_8 \wedge \delta_{35} \to \delta_9 \wedge \delta_{20} \to \delta_{36} \wedge \delta_{21} \to \delta_{37} \wedge \delta_{22} \to \delta_{38} \end{aligned}$$

Note that this has spurious consequences such as $\delta_8 \leftrightarrow \delta_9$. However, for the result, we are only interested in the freshly introduced $\delta_{33} \ldots \delta_{38}$, so spurious consequences are removed. After existential quantification we obtain the (weaker) result (changes relative to Example 5 are underlined):

$$\begin{aligned} \mathsf{g} : \forall \bar{\delta}.\, & \delta_{33} \to \delta_{38} \wedge \delta_{34} \to \delta_{36} \wedge \delta_{35} \to \delta_{37} \underline{\wedge\ \delta_{34} \to \delta_{37} \wedge \delta_{35} \to \delta_{36}} \\ & \Rightarrow \delta_{33} \mapsto \delta_{34} \mapsto \delta_{35} \mapsto (\delta_{36}, \delta_{37})^{\delta_{38}} \end{aligned}$$

Theorem 4. *The FixC rule is sound.*

8 Discussion

We have presented a binding-time analysis with a wide scope. The analysis is polyvariant and extends Dussart, Henglein and Mossin's analysis [7] to polymorphically typed programs. It applies to a functional programming language with ML-style polymorphism. The handling of (type) polymorphic application is not straightforward. We have outlined some options, utilising the increased expressiveness that we obtain by using Boolean constraints.

Types provide useful information about a program's properties and can readily be extended for various program analyses for higher-order functional languages. A recent trend is to present (and implement) program analysis as "non-standard type inference", by marrying a type language with notation for decorating type expressions, the decorations expressing the program properties of interest. The direction that much of this research is currently taking is to extend the underlying type language beyond what the programming language requires, for example to include intersection types or subtyping.

Our view is that, while it is both convenient and elegant to express program analysis as constrained-type inference, the language for expressing program properties should not necessarily be coupled closely with the type system. From the point of view of the analysis designer and implementor, it seems more attractive to utilise expressive constraint languages that come with well-developed solvers and well-understood theories, making only the reasonable assumption that programs presented to the analyser are well-typed and explicitly typed, for example by an earlier type inference phase in a compiler.

We believe that propositional logic is under-utilised in the analysis of functional programs. It offers straightforward means for expressing dependencies and disjunctive information.[2] Indeed, some of our proposed solutions to the problem of analysis in the presence of polymorphic application are expressed through the use of non-trivial propositional formulas such as $(\beta \to \beta') \to C$, as seen in Example 7.

We have a prototype implementation of the binding-time analysis presented here for the Haskell compiler, GHC. Our implementation takes a Haskell program

[2] Dussart, Henglein and Mossin [7] have disjunction in their language of binding-time properties, but only as a technical aid for the elimination of variables (as they do not have variable projection).

and assigns a binding-time type to every sub-expression. The information could be used by a program specialiser, for example Henglein and Mossin's [14] suitably extended to support type polymorphic programs. However, we have no such specialiser, and our interest is primarily the *analysis* of functional languages.

The implementation includes necessary extensions to the analysis described here so that the language accepted by GHC is supported. In particular, it supports algebraic data types, and corresponding constructors and case statements.

GHC translates the input program into a de-sugared, typed internal language, Core [17]. A GHC analysis or optimisation pass is a plug-in module that transforms an input Core program into a transformed Core program (maintaining type correctness). Since type information is explicit in the Core program the implementation is a direct translation of the analysis here. Our BTA pass annotates variables with their BTA properties. An increasing number of modern compilers have a similar internal language [23, 24] and it should not be difficult to incorporate our implementation within these.

We have experimented with two constraint solvers, Schachte's ROBDD library [22] for Boolean constraints and ImpSolver, a straightforward solver that is limited to conjunctions of implications between variables. ImpSolver is sufficient for this analysis if we employ the Test-Add method for polymorphic application. A third solver, for propositional Horn clauses, is under development for other functional program analyses.

We have run the ImpSolver based analyser on the NoFib suite of benchmark programs which is available with GHC. This suite consists of over 60 (multi-module) programs. For the 68 larger modules, those that take over 2 seconds to compile on our lightly loaded server, the average cost of the analysis is 23% of compile time, with 16% median, 1% minimum and 80% maximum. This is before non-trivial performance tuning has been attempted.

The implementation is at an early stage of evaluation. It does not yet support cross-module analysis (instead we make a 'good' guess for imported variables). Adding this is straightforward: with each exported binding in the module's interface file we add its binding-time type and list the implications which hold amongst the binding-time variables. When a binding is imported, we recreate the corresponding constraints and add the binding to the initial environment.

We believe that the ideas presented here have many immediate applications. Binding-time analysis is essentially identifying *dependencies* amongst components of expressions, and this is also the heart of several other analyses. For example, this is the case for security analysis [12, 20, 25] and for useless-code analysis [5, 15, 26], program slice analysis etc. Several researchers have addressed the problem of how to systematically extend a type system with dependency information [1, 9, 18, 19]. The goal of these researchers is similar to ours, but generally more limited with respect to scope.

Fähndrich and Rehof have recently proposed a new method for flow analysis [8]. In the absence of recursive types, the method offers improved worst-case time complexity for flow analysis, namely cubic time analysis. The key to the improvement is a concept of "instantiation" constraints. This is an extension of the constraint language which offers a way of avoiding the copying of constraints that otherwise happens when a polymorphic type is instantiated. The constraint solving problem is then translated to one of CFL reachability. A different kind of extension of the constraint language is proposed by Gustavsson and Svenningsson [11]. A "let" construct allows for "constraint abstractions", so that instantiation can be expressed directly in the constraint language. Again this leads to a cubic-time algorithm. The authors point out that the idea of incorporating instantiation into a constraint language goes back at least to Henglein's use of semi-unification constraints [13]. We are currently investigating these methods and hope to explore whether, for our implementations, the improved complexity will translate into faster analysis.

We are now developing a generic framework for specifying and implementing program analyses, based on constrained-type inference. We see this binding-time analysis as a first instance, that is, a proof of concept. We are currently applying the framework to strictness and flow analysis.

References

1. M. Abadi, A. Banerjee, N. Heintze, and J. Riecke. A core calculus of dependency. In *Proc. 26th ACM Symp. Principles of Programming Languages*, pages 147–160. ACM Press, 1999.
2. T. Armstrong, K. Marriott, P. Schachte, and H. Søndergaard. Two classes of Boolean functions for dependency analysis. *Science of Computer Programming*, 31(1):3–45, 1998.
3. K. Brace, R. Rudell, and R. Bryant. Efficient implementation of a BDD package. In *Proc. 27th ACM/IEEE Design Automation Conf.*, pages 40–45. IEEE Computer Society Press, 1990.
4. L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proc. Ninth ACM Symp. Principles of Programming Languages*, pages 207–212. ACM Press, 1982.
5. F. Damiani. *Non-Standard Type Inference for Functional Programs.* PhD thesis, Universitá di Torino, February 1998.
6. W. Dowling and J. Gallier. Linear-time algorithms for testing the satisfiability of propositional Horn formulae. *Journal of Logic Programming*, 1(3):267–284, 1984.
7. D. Dussart, F. Henglein, and C. Mossin. Polymorphic recursion and subtype qualifications: Polymorphic binding-time analysis in polynomial time. In A. Mycroft, editor, *Proc. Second Int. Symp. Static Analysis*, volume 983 of *LNCS*, pages 118–135. Springer-Verlag, 1995.
8. M. Fähndrich and J. Rehof. Type-based flow analysis: From polymorphic subtyping to CFL reachability. In *Proc. Twenty-Eighth ACM Symp. Principles of Programming Languages.* ACM Press, 2001.
9. J. S. Foster, M. Fähndrich, and A. Aiken. A theory of type qualifiers. In *Proc. 1999 ACM SIGPLAN Conf. Programming Language Design and Implementation, (SIGPLAN Notices 34(5))*, pages 192–203. ACM Press, 1999.
10. K. Glynn, P. J. Stuckey, M. Sulzmann, and H. Søndergaard. Boolean constraints for binding-time analysis. Technical Report TR2000/14, Dept. of Computer Science and Software Engineering, The University of Melbourne, 2000.

11. J. Gustavsson and J. Svenningsson. Constraint abstractions. *Proc. Second Symp. Programs as Data Objects (PADO II)*, Aarhus, Denmark, May 2001.
12. N. Heintze and J. Riecke. The SLam calculus: Programming with secrecy and integrity. In *Proc. 25th ACM Symp. Principles of Programming Languages*, pages 365–377. ACM Press, 1998.
13. F. Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, 1993.
14. F. Henglein and C. Mossin. Polymorphic binding-time analysis. In D. Sannella, editor, *Proc. European Symp. Programming (ESOP 94)*, volume 788 of *LNCS*, pages 287–301. Springer-Verlag, 1994.
15. N. Kobayashi. Type-based useless variable elimination. In *Proc. 2000 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (SIGPLAN Notices 34(11))*, pages 84–93. ACM Press, 2000.
16. T. Mogensen. Binding time analysis for polymorphically typed higher order languages. In J. Diaz and F. Orejas, editors, *Proc. Int. Conf. Theory and Practice of Software Development*, volume 352 of *LNCS*, pages 298–312. Springer-Verlag, 1989.
17. S. L. Peyton Jones and A. Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32(1-3):3–47, 1998.
18. F. Pottier and S. Conchon. Information flow inference for free. In *Proc. 5th Int. Conf. Functional Programming (ICFP'00)*, pages 46–57. ACM Press, 2000.
19. F. Prost. A static calculus of dependencies for the λ-cube. In *Proc. 15th IEEE Ann. Symp. Logic in Computer Science (LICS' 2000)*, pages 267–278. IEEE Computer Society Press, 2000.
20. A. Sabelfeld and D. Sands. A PER model of secure information flow in sequential programs. In S. Swierstra, editor, *Proc. European Symp. Programming (ESOP 99)*, volume 1576 of *LNCS*, pages 40–58. Springer, 1999.
21. V. Saraswat, M. Rinard, and P. Panangaden. Semantic foundations of concurrent constraint programming. In *Proc. Eighteenth ACM Symp. Principles of Programming Languages*, pages 333–352. ACM Press, 1991.
22. P. Schachte. Efficient ROBDD operations for program analysis. *Australian Computer Science Communications*, 18(1):347–356, 1996.
23. Z. Shao. An overview of the FLINT/ML compiler. In *Proc. 1997 ACM SIGPLAN Workshop on Types in Compilation (TIC'97)*, Amsterdam, The Netherlands, June 1997.
24. D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *Proc. ACM SIGPLAN '96 Conf. Programming Language Design and Implementation (SIGPLAN Notices 31(5))*, pages 181–192. ACM Press, 1996.
25. D. Volpano and G. Smith. A type-based approach to program security. In M. Bidoit and M. Dauchet, editors, *Theory and Practice of Software Development (TAPSOFT'97)*, volume 1214 of *LNCS*, pages 607–621. Springer, 1997.
26. M. Wand and L. Siveroni. Constraint systems for useless variable elimination. In *Proc. 26th ACM Symp. Principles of Programming Languages*, pages 291–302. ACM Press, 1999.
27. K. Wansbrough and S. L. Peyton Jones. Once upon a polymorphic type. In *Proc. 26th ACM Symp. Principles of Programming Languages*, pages 15–28. ACM Press, 1999.
28. K. Wansbrough and S. L. Peyton Jones. Simple usage polymorphism. In *Third ACM SIGPLAN Workshop on Types in Compilation (TIC 2000)*, 2000.