Inference of polymorphic and conditional strictness properties

Thomas P. Jensen
IRISA/CNRS
Campus de Beaulieu
F-35042 Rennes, France
Email: jensen@irisa.fr

Abstract

We define an inference system for modular strictness analysis of functional programs by extending a conjunctive strictness logic with polymorphic and conditional properties. This extended set of properties is used to define a syntax-directed, polymorphic strictness analysis based on polymorphic recursion whose soundness is established via a translation from the polymorphic system into the conjunctive system. From the polymorphic analysis, an inference algorithm based on constraint resolution is derived and shown complete for variant of the polymorphic analysis. The algorithm deduces at the same time a property and a set of hypotheses on the free variables of an expression which makes it suitable for analysis of program with module structure.

1 Introduction

Static analysis of program fragments with free variables usually either requires an environment associating a property with each free variable or assumes that only trivial properties hold of the free variables. For programs structured into modules the latter approach is often the only one feasible and leads to poor analysis results. This paper suggests an improvement to this by proposing a modular program analysis framework where analysis of a program fragment results in a property *and* and environment that together describes the relation between the properties of the free variables and the result of the program.

The presentation of program analyses via a set of logical inference rules (also called "non-standard type systems") is now a commonly used tool and has been applied to strictness, binding-time and control-flow analysis amongst others. Our approach consists in extending existing monomorphic analyses with polymorphic and conditional program properties (defined below). We focus on strictness analysis in this paper but a similar analysis for binding-time could be obtained from the analysis described in [10] and recently Banerjee [3] has presented a control-flow analysis for the pure lambda calculus, engineered using similar methods.

Strictness types were considered by Kuo and Mishra [21]. With $\mathtt{f}$ standing for the property "undefined", a function is strict if it maps undefined to undefined, expressed by the type $\mathtt{f} \to \mathtt{f}$. For a function of two arguments, strictness in the first argument is expressed by $\mathtt{f} \to \mathtt{t} \to \mathtt{f}$ ($\mathtt{t}$ is the trivial property "can be anything") and similarly $\mathtt{t} \to \mathtt{f} \to \mathtt{f}$ is strictness in the second argument. But we cannot express that the result of a function of two arguments is undefined as soon as one of the arguments is (as *e.g.* $+$). Conjunctive strictness types [4, 15] were introduced to combine the two properties from above into $\mathtt{f} \to \mathtt{t} \to \mathtt{f} \ \wedge \ \mathtt{t} \to \mathtt{f} \to \mathtt{f}$.

Adding conjunctions to a strictness logic increases the power of the analysis considerably; it now becomes equivalent to the abstract interpretation of Burn, Hankin and Abramsky [5]. However, contrary to the abstract interpretation, the logic is not directly suggestive of an algorithm for deducing properties of programs. Hankin and Le Métayer [11, 12] have presented an algorithm for *checking* that a program has a particular conjunctive strictness property. Monsuez [25] has developed an iteration-based algorithm for calculating polymorphic strictness properties but restricted to top-level conjunctions. In this paper we address the problem of finding an algorithm for *inferring* these properties in general.

By adding universally quantified property variables to a language of binding-time properties (without conjunctions), Dussart *et al.* have constructed a polymorphic binding-time analysis based on polymorphic recursion for analysing higher-order programs [10]. The combination of universal quantification and conjunctions yields compact representations of strictness properties: the strictness property from above is now subsumed by the property

$$\forall \alpha_1 \alpha_2 . \alpha_1 \to \alpha_2 \to (\alpha_1 \wedge \alpha_2)$$

from which all strictness properties of addition can be derived. In addition, this kind of strictness properties can be combined with the strong induction principle of polymorphic recursion to obtain a precise analysis of recursively defined functions. Due to their similarity with the polymorphic types encountered in parametric polymorphism, we call this *polymorphic strictness properties*.

In order to obtain a syntax-directed rule for the conditional construct we introduce conditional properties [28, 2] of the form $\varphi ? \alpha$, which is equivalent to φ if α becomes the value $\mathtt{t}$ but becomes $\mathtt{f}$ if α does. These kinds of properties are needed to analyse functions like $\lambda x. \mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$ which will get the type $\forall \alpha . (\alpha \to (\varphi ? \alpha))$ recording that the result is undefined if the value of x is, regardless of the property φ describing the value of the expression $\mathsf{if}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$.

To appear in the Proceedings of the 25th ACM Symposium on Principles of Programming Languages, San Diego, CA, January 1998, ACM Press

Polymorphic strictness properties are inferred by an algorithm based on Jim's work on rank-2 intersection types [20]. The algorithm is designed so that for an expression e it infers a strictness property, a set of assumptions on the free variables of e *and* a set of constraints dictating how the variables appearing in the properties can be instantiated. In order to simplify these constraints we have to extend the usual techniques of simplification to take into account the more complicated entailment relation between the polymorphic strictness properties.

In Section 2 we review conjunctive strictness logic. The language of conjunctive strictness properties is extended with polymorphic and conditional properties in Section 3 where we also show how the new properties can be translated back to pure conjunctive properties. This translation is used in Section 4 where we define a polymorphic strictness analysis and prove it correct with respect to the conjunctive logic. In Section 5 we provide a constraint-based algorithm for inferring strictness properties, prove it complete with respect to a slightly restricted version of the polymorphic strictness analysis and discuss techniques for solving the constraints generated during analysis. Section 6 relates our approach to other works in the literature and Section 7 concludes with a summary and a discussion of possible extensions.

2 Conjunctive strictness properties

In this section we define our programming language and review the conjunctive strictness logic used as a basis for the polymorphic system to be developed in the next section. The programming language studied here is the simply typed lambda calculus with integer and boolean constants and operations (+,∗,=), a conditional construct if and a fixed point operator fix. The set of types and terms of the language, Λ_T, is given by the grammar:

$$\begin{array}{rcl} \sigma & = & \mathit{Int} \mid \mathit{Bool} \mid \sigma_1 \to \sigma_2 \\ e & = & x^\sigma \mid c \mid e_1 \; op \; e_2 \mid \lambda x^\sigma . e \mid e_1 e_2 \\ & \mid & \mathsf{fix}(\lambda f.e) \mid \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3. \end{array}$$

where *Int* is the type of integers and *Bool* that of booleans. We assume that every variable appearing in the program has been annotated with the type of values it ranges over but we leave it out where we can get away with it. For each type σ we define a set of conjunctive program properties $L_\wedge(\sigma)$ as follows:

$$\mathtt{t}, \mathtt{f} \in L_\wedge(\sigma) \qquad \frac{\varphi_1 \in L_\wedge(\sigma_1) \quad \varphi_2 \in L_\wedge(\sigma_2)}{\varphi_1 \to \varphi_2 \in L_\wedge(\sigma_1 \to \sigma_2)}$$

$$\frac{\varphi_i \in L_\wedge(\sigma) \quad i \in I}{\bigwedge_{i\in I} \varphi_i \in L_\wedge(\sigma)}$$

where I is a finite, non-empty set. We write φ^σ to indicate the type of a property. The property $\mathtt{t}^\sigma$ is the trivial property that holds for all values of type σ. The interpretation of $\mathtt{f}$ is "the value of this expression is undefined". In strictness analysis, $\mathtt{f} \to \mathtt{f}$ means that the function maps an undefined argument to an undefined result *i.e.* that the function is strict. Similarly, $\mathtt{t} \to \mathtt{f}$ means that all values are mapped to undefined.

On each set $L_\wedge(\sigma)$ of properties is defined an implication ordering $\leq$ that extends the basic implication $\mathtt{f}^\sigma \leq \mathtt{t}^\sigma$ to conjunctions and function properties. We define $=$ to mean that two properties φ and ψ are provably equivalent *i.e.* that $\varphi \leq \psi$ and $\psi \leq \varphi$. The rules defining $\leq$ are given in Figure 1.

- $\dfrac{\varphi \leq \psi_i, \quad i \in I}{\varphi \leq \bigwedge_{i\in I} \psi_i}$
- $\bigwedge_{i\in I} \varphi_i \leq \varphi_i, \quad \forall i \in I$
- $\varphi \leq \mathtt{t}$
- $\mathtt{t}^{\sigma\to\tau} = \mathtt{t}^\sigma \to \mathtt{t}^\tau$
- $\mathtt{f} \leq \varphi$
- $\mathtt{t}^\sigma \to \mathtt{f}^\tau = \mathtt{f}^{\sigma\to\tau}$
- $\bigwedge_{i\in I}(\varphi \to \psi_i) = \varphi \to \bigwedge_{i\in I} \psi_i$
- $\dfrac{\psi_1 \leq \varphi_1, \varphi_2 \leq \psi_2}{\varphi_1 \to \varphi_2 \leq \psi_1 \to \psi_2}$

Figure 1: Axiomatisation of conjunctive strictness properties

The following predicate identifies the set of *irreducible* formulae *i.e.* those formulae that cannot be expressed as a conjunction of strictly weaker formulae.

$$\mathsf{Irr}(\mathtt{f}^\sigma) \qquad \frac{\forall i \in I. \mathsf{Irr}(\psi_i) \quad \mathsf{Irr}(\psi)}{\mathsf{Irr}((\bigwedge_{i\in I} \psi_i) \to \psi)}. \qquad \mathsf{Irr}(\mathtt{t}^\sigma)$$

Proposition 2.1 *For all $\psi \in L(\sigma)$ there exist a finite set $\{\psi_i\}_{i\in I}$ with $\mathsf{Irr}(\psi_i)$ such that $\psi = \bigwedge_{i\in I} \psi_i$.*

The proposition is used to establish the main result [15, 18] about this axiomatisation: when quotienting the preorder $(L_\wedge(\sigma), \leq)$ by provable equivalence we obtain a finite lattice of properties and this lattice is exactly the lattice used in the strictness analysis for higher-order functional languages defined by Burn, Hankin and Abramsky [5]. In other words, for each type we can exhibit a finite set $L_{NF}(\sigma)$ of "normal forms" such that to each formula φ there exists a normal form $[\varphi]$ satisfying $\varphi = [\varphi]$. For the base types *Int* and *Bool*, the set of normal forms is the two-point lattice since all formulae are equivalent to either $\mathtt{t}$ or $\mathtt{f}$. We assume that the formulae $\mathtt{t}$ and $\mathtt{f}$ are chosen as normal forms.

2.1 The strictness logic

The set of conjunctive strictness formulae forms the basis of a logic for reasoning about strictness properties of functions. It has been shown [15, 18] that the strongest property provable of a program in the logic corresponds exactly to the property that the abstract interpretation cited above will find. In this sense, the two analyses are equivalent. The logic is defined by the inference rules in Figure 2. A judgment is of form $\Delta \vdash_\wedge e : \phi$ where e is an expression, ϕ is a formula describing a property of e and Δ is an environment associating program variables in the expression e with properties. The logic differs in presentation from the original conjunctive strictness logic [4, 15] by having inequalities as side conditions on some of the inference rules and in its treatment of base operators. These changes reduce the need for the rule **Weak**: it can be shown that any sequent provable in the logic can be proved by a deduction where the **Weak**-rule is only used as the last step in the deduction. We notice that the **Weak**-rule only allows weakening of the conclusion ϕ and not strengthening of the hypothesis Δ. However, it can be shown that if $\Delta' \leq \Delta$ and $\Delta \vdash e : \phi$ then we can obtain a proof of $\Delta' \vdash e : \phi$ by inserting weakenings after look-ups in the environment with the **Var**-rule.

$$\textbf{Taut}\quad \frac{FV(e) \subseteq Dom(\Delta)}{\Delta \vdash_\wedge e : \mathtt{t}} \qquad \textbf{Var}\quad \Delta[x \mapsto \phi] \vdash_\wedge x : \phi$$

$$\textbf{Conj}\quad \frac{\Delta \vdash_\wedge e : \psi_1 \quad \Delta \vdash_\wedge e : \psi_2}{\Delta \vdash_\wedge e : \psi_1 \wedge \psi_2}$$

$$\textbf{Weak}\quad \frac{\Delta \vdash_\wedge e : \phi \quad \phi \leq \psi}{\Delta \vdash_\wedge e : \psi}$$

$$\textbf{Abs}\quad \frac{\Delta[x \mapsto \phi] \vdash_\wedge e : \psi}{\Delta \vdash_\wedge \lambda x.e : (\phi \to \psi)}$$

$$\textbf{Fix}\quad \frac{\Delta[f : \psi_1] \vdash_\wedge e : \psi_2 \quad \psi_2 \leq \psi_1}{\Delta \vdash_\wedge \mathsf{fix}(\lambda f.e) : \psi_2}$$

$$\textbf{App}\quad \frac{\Delta \vdash_\wedge e_1 : (\phi_1 \to \phi_2) \quad \Delta \vdash_\wedge e_2 : \psi \quad \psi \leq \phi_1}{\Delta \vdash_\wedge e_1 e_2 : \phi_2}$$

$$\textbf{Base-L}\quad \frac{\Delta \vdash_\wedge e_1 : \varphi_1 \quad \Delta \vdash_\wedge e_2 : \varphi_2}{\Delta \vdash_\wedge e_1 \; op \; e_2 : \varphi_1}$$

$$\textbf{Base-R}\quad \frac{\Delta \vdash_\wedge e_1 : \varphi_1 \quad \Delta \vdash_\wedge e_2 : \varphi_2}{\Delta \vdash_\wedge e_1 \; op \; e_2 : \varphi_2}$$

$$\textbf{If-1}\quad \frac{FV(e_1) \cup FV(e_2) \subseteq Dom(\Delta) \quad \Delta \vdash_\wedge b : \mathtt{f}}{\Delta \vdash_\wedge \mathsf{if}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \mathtt{f}}$$

$$\textbf{If-2}\quad \frac{FV(b) \subseteq Dom(\Delta) \quad \Delta \vdash_\wedge e_1 : \phi \quad \Delta \vdash_\wedge e_2 : \phi}{\Delta \vdash_\wedge \mathsf{if}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \phi}$$

Figure 2: Conjunctive strictness logic

3 Polymorphic and conditional program properties

The conjunctive program properties are expressive enough to define all properties used in the abstract interpretation of higher-order functions [5] based on monotone functions between lattices. This is achieved through the use of conjunctions that allows us to list all properties of a given function. However, it can be cumbersome for an analyser to work with such conjunctions that tend to grow as more is known about a program. In this section we extend the language of properties in two ways: for each type we add property variables and define a notion of property scheme and we introduce two new property constructors, ? and $\Leftarrow$, for building conditional properties. These features allow us to express program properties succinctly and leads to a constraint-based algorithm for inferring such properties. Both features are inspired by similar constructs for typing of programs [24, 2].

The two base types *Int* and *Bool* have isomorphic sets of properties and it turns out to be convenient to identify these two sets. We do this by replacing *Int* and *Bool* with a common base type $\flat$ in the definition of the set of properties. A property of the set $L(\flat)$ can describe both an integer and a boolean value. Formally, we redefine the set of types to be

$$\sigma = \flat \mid \sigma_1 \to \sigma_2$$

Assume that for each type σ we are given a set of property variables, ranged over by $\alpha^\sigma, \beta^\sigma$. We extend the set of properties by defining the type-indexed collection of properties $L(\sigma)$ as follows:

- $\varphi^\sigma ? \mathtt{f}^\flat = \mathtt{f}^\sigma$ • $\varphi^\sigma ? \mathtt{t}^\flat = \varphi^\sigma$
- $\varphi^\sigma \Leftarrow \mathtt{f}^\flat = \mathtt{t}^\sigma$ • $\varphi^\sigma \Leftarrow \mathtt{t}^\flat = \varphi^\sigma$
- $(\varphi \Leftarrow \varphi_1^\flat) \Leftarrow \varphi_2^\flat = \varphi \Leftarrow (\varphi_1^\flat \wedge \varphi_2^\flat)$
- $(\varphi ? \varphi_1^\flat) ? \varphi_2^\flat = \varphi ? (\varphi_1^\flat \wedge \varphi_2^\flat)$
- $(\varphi_1 \wedge \varphi_2) ? \varphi = (\varphi_1 ? \varphi) \wedge (\varphi_2 ? \varphi)$
- $(\varphi_1 \wedge \varphi_2) \Leftarrow \varphi = (\varphi_1 \Leftarrow \varphi) \wedge (\varphi_2 \Leftarrow \varphi)$
- $(\varphi_1 \to \varphi_2) ? \varphi = (\varphi_1 \Rightarrow \varphi) \to (\varphi_2 ? \varphi)$
- $(\varphi_1 \to \varphi_2) \Leftarrow \varphi = (\varphi_1 \Rightarrow \varphi) \to (\varphi_2 \Leftarrow \varphi)$
- $\dfrac{\varphi_1 \leq \psi_1 \quad \varphi_2 \leq \psi_2}{\varphi_2 ? \varphi_1 \leq \psi_2 ? \psi_1}$ • $\dfrac{\psi_1 \leq \varphi_1 \quad \varphi_2 \leq \psi_2}{\varphi_2 \Leftarrow \varphi_1 \leq \psi_2 \Leftarrow \psi_1}$
- $\forall \alpha^\sigma . \varphi \leq \varphi[\psi/\alpha^\sigma] \quad , \quad \psi \in L(\sigma)$
- $\dfrac{\varphi \leq \psi \quad \alpha \notin FV(\psi)}{\varphi \leq \forall \alpha . \psi}$

Figure 3: Rules for polymorphic properties.

Definition 3.1 *For each type σ define $L(\sigma)$ to be the smallest set satisfying*

$$\mathtt{t}, \mathtt{f}, \alpha^\sigma \in L(\sigma) \qquad \frac{\varphi_1 \in L(\sigma_1) \quad \varphi_2 \in L(\sigma_2)}{\varphi_1 \to \varphi_2 \in L(\sigma_1 \to \sigma_2)}$$

$$\frac{\varphi_i \in L(\sigma) \quad i \in I}{\bigwedge_{i \in I} \varphi_i \in L(\sigma)} \qquad \frac{\varphi \in L(\sigma)}{\forall \alpha . \varphi \in L(\sigma)}$$

$$\frac{\varphi_2 \in L(\flat) \quad \varphi_1 \in L(\sigma)}{\varphi_1 ? \varphi_2 \in L(\sigma)}. \qquad \frac{\varphi_2 \in L(\flat) \quad \varphi_1 \in L(\sigma)}{\varphi_1 \Leftarrow \varphi_2 \in L(\sigma)}.$$

The variables α^σ range over the properties of type σ. As an example, we have that the identity function on integers satisfies $\forall \alpha^\flat . \alpha^\flat \to \alpha^\flat$ from which we get as an instance that the identity satisfies $\mathtt{f} \to \mathtt{f}$ *i.e.* that it is strict. As a somewhat more interesting example that mixes polymorphism and conjunctions we have the addition operator that is bi-strict: as soon as one argument is undefined the result is undefined. This is expressed by the property

$$\forall \alpha_1^\flat \alpha_2^\flat . \alpha_1^\flat \to \alpha_2^\flat \to \alpha_1^\flat \wedge \alpha_2^\flat .$$

The notions of free and bound variables of a property φ (written $FV(\varphi)$ and $BV(\varphi)$) and renaming of bound variables carry over to properties in the standard way. We do not distinguish between properties that only differ in the choice of bound variables. Substitution of ψ for all free occurrences of variable α in φ is written $\varphi[\psi/\alpha]$.

Conditional properties have the form $\varphi_1 ? \varphi_2$ where φ_2 is a property of base type. They were used for dataflow analysis by Reynolds [28] and for soft (or partial) typing of functional programs by Aiken *et al.* [2]. If $\varphi_2 = \mathtt{f}^\flat$ then $\varphi_1 ? \varphi_2$ is equal to $\mathtt{f}^\sigma$ and if $\varphi_2 = \mathtt{t}^\flat$ then $\varphi_1 ? \varphi_2$ is equal to φ_1 where $\varphi \in L(\sigma)$. This functionality could conveniently have been expressed as the conjunction of φ_1 and φ_2, were it not for the fact that φ_1 and φ_2 have different types. We need this "conjunction across types" to state that the result of an if-statement of type σ is undefined (satisfies $\mathtt{f}^\sigma$) if the boolean condition in the expression is undefined (satisfies $\mathtt{f}^\flat$). The $\Leftarrow$ is the dual of ? and behaves like implication across types in that $\varphi_2 \Leftarrow \mathtt{t}^\flat = \varphi_2$ and $\varphi_2 \Leftarrow \mathtt{f}^\flat = \mathtt{t}^\sigma$. The constructor $\Leftarrow$ does not appear in the inference rules in Figure 4 defining the logic; it serves to reduce conditional properties at higher types since the equalities

$$(\varphi_1 \to \varphi_2)?\psi = (\varphi_1 \Leftarrow \psi) \to (\varphi_2?\psi)$$

and

$$(\varphi_1 \to \varphi_2) \Leftarrow \psi = (\varphi_1 \Leftarrow \psi) \to (\varphi_2 \Leftarrow \psi).$$

are semantically valid (Lemma 3.4). For example, we have that

$$(\mathtt{f} \to \mathtt{f})?\mathtt{f} = (\mathtt{f} \Leftarrow \mathtt{f}) \to (\mathtt{f}?\mathtt{f}) = (\mathtt{t} \to \mathtt{f}).$$

The upshot of this is that we only have to consider conditional properties at base types when we want to decide whether one property entails another (see Section 5). The rules in Figure 3 extend the implication ordering to polymorphic and conditional properties.

3.1 Translation into conjunctive properties

As a step towards relating the new properties in $L(\sigma)$ to those in $L_\wedge(\sigma)$ we define a translation, $\_^\circ$, that removes quantifiers and satisfies that for a closed property $\varphi \in L(\sigma)$ (*i.e.* a property with no free variables) we have that $\varphi^\circ \in L_\wedge(\sigma)$. The key to the translation is the fact from the end of Section 2 that there is only a finite number of different properties at each type and that the α^σ in $\forall\alpha^\sigma.\varphi$ therefore only ranges over a finite number of different properties. This means that a quantified type can be translated into a finite conjunction.

Definition 3.2 *The translation* $\_^\circ : L(\sigma) \to L(\sigma)$ *is defined by*

$$\begin{array}{lcl}
(\forall\alpha^\sigma.\varphi)^\circ & = & \bigwedge_{\psi\in L_{(NF)}(\sigma)}(\varphi[\psi/\alpha^\sigma])^\circ \\
(\varphi_1?\varphi_2)^\circ & = & \left\{\begin{array}{ll} \varphi_1^\circ & \textit{if } [\varphi_2^\circ] = \mathtt{t}^\flat \\ \mathtt{f}^\sigma & \textit{if } [\varphi_2^\circ] = \mathtt{f}^\flat \\ \varphi_1^\circ?\varphi_2^\circ & \textit{otherwise} \end{array}\right. \\
(\varphi_1 \to \varphi_2)^\circ & = & \varphi_1^\circ \to \varphi_2^\circ \\
\alpha^\circ & = & \alpha \\
(\bigwedge_{i\in I}\varphi_i)^\circ & = & \bigwedge_{i\in I}(\varphi_i^\circ) \\
(\varphi_1 \Leftarrow \varphi_2)^\circ & = & \left\{\begin{array}{ll} \varphi_1^\circ & \textit{if } [\varphi_2^\circ] = \mathtt{t}^\flat \\ \mathtt{t}^\sigma & \textit{if } [\varphi_2^\circ] = \mathtt{f}^\flat \\ \varphi_1^\circ \Leftarrow \varphi_2^\circ & \textit{otherwise} \end{array}\right. \\
\mathtt{t}^\circ & = & \mathtt{t} \\
\mathtt{f}^\circ & = & \mathtt{f}
\end{array}$$

The translation leaves the set of free variables of a property unchanged. These disappear when a *grounding* substitution is applied to the property.

Definition 3.3 *A substitution S is a finite map from property variables to properties such that $S(\alpha^\sigma) \in L(\sigma)$. Its domain is denoted $Dom(S)$. S is* ground *if for all $\alpha^\sigma \in Dom(S)$, we have $S(\alpha^\sigma) \in L_\wedge(\sigma)$. A ground substitution S is said to* ground φ *if $FV(\varphi) \subseteq Dom(S)$.*

Thus, for a ground substitution S, $S(\varphi^\circ) \in L_\wedge(\sigma)$ if $\varphi \in L(\sigma)$. Furthermore, translation commutes with substitution:

Lemma 3.4 *For all S grounding φ, ψ, we have*

$$S(\varphi^\circ) = (S(\varphi))^\circ$$

and

$$\varphi \leq \psi \quad \Rightarrow \quad S(\varphi^\circ) \leq S(\psi^\circ)$$

Proof (sketch): The first part follows from the fact that the translation leaves the set of free variables of a term invariant. The proof of the second part consists of checking each rule defining the ordering $\leq$. For example, the rule for instantiation

$$\forall\alpha^\sigma.\varphi \leq \varphi[\psi/\alpha^\sigma] \quad , \quad \psi \in L(\sigma)$$

which means that we have to show

$$S(\forall\alpha^\sigma.\varphi)^\circ = \bigwedge_{\psi\in L_{(NF)}(\sigma)} S(\varphi^\circ[\psi/\alpha^\sigma]) \leq S(\varphi)^\circ[S(\psi)^\circ/\alpha^\sigma]$$

Now, $S(\psi)^\circ$ is quantifier-free and without free variables so all conditional properties can be reduced away. Therefore there exists a $\psi \in L_{(NF)}(\sigma)$ equivalent to $S(\psi)^\circ$ and the inequality follows.

Equalities like

$$S((\varphi_1 \to \varphi_2)?\psi)^\circ = S((\varphi_1 \Leftarrow \psi) \to (\varphi_2?\psi))^\circ$$

are established by inspecting each of the cases for $S(\psi) = \mathtt{t}^\flat$ and $S(\psi) = \mathtt{f}^\flat$ □

4 Polymorphic strictness analysis

The polymorphic strictness analysis defined in this section operates on judgments of the form $\Gamma \vdash e : \varphi$ where the environment Γ is a finite set of assumptions of the form $x_i^\sigma : \varphi_i^\sigma$ associating properties to program variables. The set of assumptions Γ will always contain exactly one assumption for every free program variable in e. To combine several assumption sets into one we define $\Gamma_1 + \Gamma_2$ to be $\Gamma_1 \cup \Gamma_2$ except that if $x : \varphi_1 \in \Gamma_1$ and $x : \varphi_2 \in \Gamma_2$ then these are replaced by $x : \varphi_1 \wedge \varphi_2$ in $\Gamma_1 + \Gamma_2$.

A property variable α^σ is said to be free in Γ if it is free in one or more of the φ_i^σ. The set of free variables in Γ is denoted by $FV(\Gamma)$. For given environment Γ and property φ we define the property $Gen(\Gamma, \varphi) \equiv \forall\overline{\alpha}.\varphi$ obtained by quantifying φ over all variables free in φ and not free in Γ. Polymorphic type systems usually contain rules for introducing and eliminating quantified types, *viz.*,

$$Gen\frac{\Gamma \vdash e : \tau \quad \alpha \notin FV(\Gamma)}{\Gamma \vdash e : \forall\alpha.\tau} \qquad Int\frac{\Gamma \vdash e : \forall\alpha.\tau}{\Gamma \vdash e : \tau[\tau'/\alpha]}.$$

However, these rules breaks the syntax-directed property of the inference system and it is therefore common to modify the other rules in the system to incorporate *Gen* and *Int*. We adhere to this approach: the system that we propose here does not assign quantified types to expressions though it does use them internally.

In order to deal with recursive definitions it is important that the rule for fix allows that different occurrences of the recursive function variable f in fix $\lambda f.e$ can be ascribed different properties. In the conjunctive system this was naturally achieved by combining the different properties using conjunctions and then select the relevant property at each occurrence using the weakening rule. However, weakening breaks the syntax-directedness of a system so it must be eliminated and built into the other rules. One way of achieving this is to use the principle of polymorphic recursion proposed by Mycroft [26] for typing ML programs. This principle can be expressed by the inference rule

$$\frac{\Gamma \cup \{f : \bigwedge_{i \in I} \varphi_i\} \vdash e : \varphi \quad \forall i \in I : Gen(\Gamma, \varphi) \leq \varphi_i}{\Gamma \vdash \mathsf{fix}\ \lambda f.e : \varphi}.$$

Informally, it states that each hypothesis φ_i made on f to analyse the body e must be implied by the generalisation of the property deduced for e. Notice that we can generalise over variables that are free in φ_i as long as they are not free in Γ. For example, having proved that the body of a function satisfies $\alpha_2 \to \alpha_1 \to (\alpha_1 \wedge \alpha_2)$ under the assumption $\{f : \alpha_1 \to \alpha_2 \to (\alpha_1 \wedge \alpha_2)\}$, we can generalise over all free variables (α_1, α_2) to obtain $\forall \alpha_1 \alpha_2 . \alpha_2 \to \alpha_1 \to (\alpha_1 \wedge \alpha_2)$ of which the assumption on f is an instance. We give an example of a deduction in the system in Figure 5.

Rules

$$\{x : \varphi\} \vdash x : \varphi \qquad \emptyset \vdash c : \mathtt{t}$$

$$\frac{\Gamma \vdash e : \varphi \quad x \notin FV(e)}{\Gamma \vdash \lambda x.e : \mathtt{t} \to \varphi} \qquad \frac{\Gamma \cup \{x : \varphi\} \vdash e : \psi}{\Gamma \vdash \lambda x.e : \varphi \to \psi}$$

$$\frac{\Gamma \vdash e_1 : ((\bigwedge_{i \in I} \varphi_i) \to \varphi)?\varphi_b \quad \Gamma_i \vdash e_2 : \psi_i \quad \forall i \in I : \psi_i \leq \varphi_i}{\Gamma + \sum_{i \in I} \Gamma_i \vdash e_1 e_2 : \varphi?\varphi_b}$$

$$\frac{\Gamma_i \vdash e_i : \varphi_i \ , i = 1,2,3 \quad \varphi_2 \leq \varphi \quad \varphi_3 \leq \varphi}{\Gamma_1 + \Gamma_2 + \Gamma_3 \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \varphi?\varphi_1}$$

$$\frac{\Gamma \cup \{f : \bigwedge_{i \in I} \varphi_i\} \vdash e : \varphi \quad \forall i \in I : Gen(\Gamma, \varphi) \leq \varphi_i}{\Gamma \vdash \mathsf{fix}\ \lambda f.e : \varphi}$$

$$\frac{\Gamma_1 \vdash e_1 : \varphi_1 \quad \Gamma_2 \vdash e_2 : \varphi_2}{\Gamma_1 + \Gamma_2 \vdash e_1\ op\ e_2 : \varphi_1 \wedge \varphi_2} \qquad \frac{\Gamma \vdash e : \varphi \quad f \notin FV(e)}{\Gamma \vdash \mathsf{fix}\ \lambda f.e : \varphi}$$

Figure 4: Polymorphic strictness analysis.

As opposed to the system $\vdash_\wedge$, we now have that the variables in the environment are exactly the free variables in e; hence the need for two rules for λ-abstraction and fix. This eliminates the arbitrariness in the conjunctive system that allowed to add irrelevant properties to the environment. When the rule for basic operators *op* is instantiated to e.g. $=$ we take advantage of having a common set of properties for integer and boolean values since we otherwise would have to convert the conjunction $\varphi_1 \wedge \varphi_2$ of integer properties to the equivalent property for booleans.

The example in Figure 5 illustrates the analysis by showing the deduction of property

$$\alpha_1 \to \alpha_2 \to \alpha_3 \to ((\alpha_1 \wedge \alpha_2)?\alpha_3)$$

for the function f, recursively defined by:[1]

$$\begin{array}{l} \mathsf{fix}\ \lambda f.\lambda xyz.\mathsf{if}\ z = 0\ \mathsf{then}\ x - y\ \mathsf{else} \\ \qquad\qquad \mathsf{if}\ z < 0\ \mathsf{then}\ f\ x\ y\ (1 - z) \\ \qquad\qquad\quad \mathsf{else}\ f\ y\ x\ (z - 1). \end{array}$$

[1] More natural examples would be possible once algebraic data types are added to the language.

The function is an elaboration of an example due to Kuo and Mishra [21]. The function f is strict in each of the arguments x and y separately *i.e.*, it satisfies the property $\mathtt{f} \to \mathtt{t} \to \mathtt{t} \to \mathtt{f} \ \wedge \ \mathtt{t} \to \mathtt{f} \to \mathtt{t} \to \mathtt{f}$. In a purely conjunctive logic, two sub-deductions with the same structure are needed to prove that the function satisfies each of the conjuncts, see [18]. These sub-deductions are combined into one in our analysis by using quantified property variables and polymorphic recursion; more precisely, we prove that f satisfies

$$\alpha_1 \to \alpha_2 \to \alpha_3 \to ((\alpha_1 \wedge \alpha_2)?\alpha_3).$$

The two recursive calls to f will need two different properties of f that both are implied by the generalisation of the one above *viz.*,

$$\alpha_2 \to \alpha_1 \to \alpha_3 \to (\alpha_2 \wedge \alpha_1)$$

and

$$\alpha_1 \to \alpha_2 \to \alpha_3 \to (\alpha_1 \wedge \alpha_2).$$

4.1 Soundness

The soundness of this system is stated with respect to the conjunctive strictness logic of [4, 15] as presented in Section 2. Deductions in this logic are written with a $\vdash_\wedge$ to distinguish them from deductions in the system defined in this section.

Theorem 4.1 (Soundness) *For all S grounding φ, φ_i:*

$$\{x_i : \varphi_i\} \vdash e : \varphi \quad \Rightarrow \quad \{x_i : S(\varphi_i^\circ)\} \vdash_\wedge e : S(\varphi^\circ).$$

Proof: Proof by rule induction; we give the cases for if and fix. Assume that

$$\Gamma_1 + \Gamma_2 + \Gamma_3 \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \varphi?\varphi_1.$$

Recalling that none of the formulae involve the $\forall$-constructor (so the $(\ )^\circ$ translation is the identity), we have to show that for all grounding S,

$$S(\Gamma_1 + \Gamma_2 + \Gamma_3) \vdash_\wedge \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : S(\varphi?\varphi_1) = S(\varphi)?S(\varphi_1).$$

There are two cases to consider: if $S(\varphi_1) = \mathtt{f}^\flat$ then by induction we have that

$$S(\Gamma_1) \vdash_\wedge e_1 : S(\varphi_1) = \mathtt{f}^\flat$$

so by rule **If-1**,

$$S(\Gamma_1) \vdash_\wedge \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \mathtt{f} = S(\varphi)?S(\varphi_1)$$

Since $S(\Gamma_1 + \Gamma_2 + \Gamma_3) \leq S(\Gamma_1)$, we know that there exists a proof of

$$S(\Gamma_1 + \Gamma_2 + \Gamma_3) \vdash_\wedge \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \mathtt{f} = S(\varphi)?S(\varphi_1)$$

cf. the remark at the end of Section 2.1. Otherwise, if $S(\varphi_1) = \mathtt{t}^\flat$ then $S(\varphi)?S(\varphi_1) = S(\varphi)$ so by induction we have that

$$S(\Gamma_2) \vdash e_2 : S(\varphi) \qquad \text{and} \qquad \mathrm{S}(\Gamma_3) \vdash \mathrm{e}_3 : \mathrm{S}(\varphi)$$

and, by an argument similar to above,

$$S(\Gamma_1 + \Gamma_2 + \Gamma_3) \vdash e_2 : S(\varphi) \quad \text{and} \quad \mathrm{S}(\Gamma_1 + \Gamma_2 + \Gamma_3) \vdash \mathrm{e}_3 : \mathrm{S}(\varphi).$$

The conclusion is now obtained by using **If-2**.

Analysis of the function

$$\mathsf{fix}\ \lambda f.\lambda xyz.\mathsf{if}\ z=0\ \mathsf{then}\ x-y\ \mathsf{else}\ \mathsf{if}\ z<0\ \mathsf{then}\ f\ x\ y\ (1-z)\ \mathsf{else}\ f\ y\ x\ (z-1).$$

Let

$\Gamma_1=\{x:\alpha_1\}, \Gamma_2=\{y:\alpha_2\}, \Gamma_3=\{z:\alpha_3\}$,
$\Gamma_4=\{f:\alpha_2\to\alpha_1\to\alpha_3\to(\alpha_1\wedge\alpha_2)\}, \Gamma_5=\{f:\alpha_1\to\alpha_2\to\alpha_3\to(\alpha_1\wedge\alpha_2)\}$ and
$\Gamma=\sum_{i=1}^{5}\Gamma_i=\{x:\alpha_1, y:\alpha_2, z:\alpha_3, f:(\alpha_2\to\alpha_1\to\alpha_3\to(\alpha_1\wedge\alpha_2))\wedge(\alpha_1\to\alpha_2\to\alpha_3\to(\alpha_1\wedge\alpha_2))\}$.

The analysis of the inner if-expression consists of combining the three deductions:

$$\frac{\Gamma_5\vdash f:\alpha_1\to\alpha_2\to\alpha_3\to(\alpha_1\wedge\alpha_2)\quad \Gamma_1\vdash x:\alpha_1\quad \Gamma_2\vdash y:\alpha_2\quad \dfrac{\emptyset\vdash 1:\mathtt{t}\quad \Gamma_3\vdash z:\alpha_3}{\Gamma_3\vdash(1-z):\alpha_3}}{\Gamma_1+\Gamma_2+\Gamma_3+\Gamma_5\vdash f\,x\,y\,(1-z):\alpha_1\wedge\alpha_2}$$

$$\frac{\Gamma_3\vdash z:\alpha_3\quad \emptyset\vdash 0:\mathtt{t}}{\Gamma_3\vdash z<0:\alpha_3}$$

and

$$\frac{\Gamma_4\vdash f:\alpha_2\to\alpha_1\to\alpha_3\to(\alpha_1\wedge\alpha_2)\quad \Gamma_2\vdash y:\alpha_2\quad \Gamma_1\vdash x:\alpha_1\quad \begin{array}{c}\vdots\\ \Gamma_3\vdash(z-1):\alpha_3\end{array}}{\Gamma_1+\Gamma_2+\Gamma_3+\Gamma_4\vdash f\,y\,x\,(z-1):\alpha_1\wedge\alpha_2}$$

in order to deduce

$$\Gamma\vdash\mathsf{if}\ z<0\ \mathsf{then}\ f\,x\,y\,(1-z)\ \mathsf{else}\ f\,y\,x\,(z-1):(\alpha_1\wedge\alpha_2)?\alpha_3$$

using the rule for if. The analysis then proceeds as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma_3\vdash z:\alpha_3\quad \emptyset\vdash 0:\mathtt{t}}{\Gamma_3\vdash z=0:\alpha_3}\quad \dfrac{\Gamma_1\vdash x:\alpha_1\quad \Gamma_2\vdash y:\alpha_2}{\Gamma_1+\Gamma_2\vdash x-y:\alpha_1\wedge\alpha_2}\quad \begin{array}{c}\vdots\\ \Gamma\vdash\mathsf{if}\ z<0\ \mathsf{then}\ f\,x\,y\,(1-z)\ \mathsf{else}\ f\,y\,x\,(z-1):(\alpha_1\wedge\alpha_2)?\alpha_3\end{array}}{\Gamma_1+\Gamma_2+\Gamma_3+\Gamma_4+\Gamma_5\vdash\mathsf{if}\ z=0\ \mathsf{then}\ x-y\ \mathsf{else}\ \mathsf{if}\ z<0\ \mathsf{then}\ f\,x\,y\,(1-z)\ \mathsf{else}\ f\,y\,x\,(z-1):(\alpha_1\wedge\alpha_2)?\alpha_3}}{\Gamma_1+\Gamma_2+\Gamma_4+\Gamma_5\vdash\lambda z.\mathsf{if}\ z=0\ \mathsf{then}\ x-y\ \mathsf{else}\ \mathsf{if}\ z<0\ \mathsf{then}\ f\,x\,y\,(1-z)\ \mathsf{else}\ f\,y\,x\,(z-1):\alpha_3\to((\alpha_1\wedge\alpha_2)?\alpha_3)}}{\Gamma_1+\Gamma_4+\Gamma_5\vdash\lambda yz.\mathsf{if}\ z=0\ \mathsf{then}\ x-y\ \mathsf{else}\ \mathsf{if}\ z<0\ \mathsf{then}\ f\,x\,y\,(1-z)\ \mathsf{else}\ f\,y\,x\,(z-1):\alpha_2\to\alpha_3\to((\alpha_1\wedge\alpha_2)?\alpha_3)}}{\Gamma_4+\Gamma_5\vdash\lambda xyz.\mathsf{if}\ z=0\ \mathsf{then}\ x-y\ \mathsf{else}\ \mathsf{if}\ z<0\ \mathsf{then}\ f\,x\,y\,(1-z)\ \mathsf{else}\ f\,y\,x\,(z-1):\alpha_1\to\alpha_2\to\alpha_3\to((\alpha_1\wedge\alpha_2)?\alpha_3)}}{\vdash\mathsf{fix}\ \lambda f.\lambda xyz.\mathsf{if}\ z=0\ \mathsf{then}\ x-y\ \mathsf{else}\ \mathsf{if}\ z<0\ \mathsf{then}\ f\,x\,y\,(1-z)\ \mathsf{else}\ f\,y\,x\,(z-1)):\alpha_1\to\alpha_2\to\alpha_3\to((\alpha_1\wedge\alpha_2)?\alpha_3)}$$

During the deduction we have used that $\alpha_3\wedge\mathtt{t}=\alpha_3$ and that $(\alpha_1\wedge\alpha_2)?\alpha_3\leq\alpha_1\wedge\alpha_2$. In the last step of the deduction it was used that by instantiating α_1 with α_2 and α_2 with α_1 we get

$$Gen(\emptyset,\alpha_1\to\alpha_2\to\alpha_3\to(\alpha_1\wedge\alpha_2)?\alpha_3)=\forall\alpha_1\alpha_2\alpha_3.\alpha_1\to\alpha_2\to\alpha_3\to((\alpha_1\wedge\alpha_2)?\alpha_3)\leq\alpha_2\to\alpha_1\to\alpha_3\to(\alpha_1\wedge\alpha_2)$$

and

$$Gen(\emptyset,\alpha_1\to\alpha_2\to\alpha_3\to(\alpha_1\wedge\alpha_2)?\alpha_3)=\forall\alpha_1\alpha_2\alpha_3.\alpha_1\to\alpha_2\to\alpha_3\to((\alpha_1\wedge\alpha_2)?\alpha_3)\leq\alpha_1\to\alpha_2\to\alpha_3\to(\alpha_1\wedge\alpha_2).$$

Figure 5: An example deduction

For the rule for recursion we assume that for all grounding substitutions S' we have

$$S'(\Gamma^\circ) \cup \{f : S'(\bigwedge_{i\in I} \varphi_i^\circ)\} \vdash_\wedge e : S'(\varphi^\circ) \qquad (*)$$

and that $Gen(\Gamma, \varphi) \leq \varphi_i$ for all $i \in I$. From this we have to prove that

$$S(\Gamma^\circ) \vdash_\wedge \mathsf{fix}\ \lambda f.e : S(Gen(\Gamma, \varphi)^\circ)$$

for all grounding substitutions S. For this it suffices to show

$$S(\Gamma^\circ) \cup \{f : S(Gen(\Gamma, \varphi)^\circ)\} \vdash_\wedge e : S(Gen(\Gamma, \varphi)^\circ).$$

The property $S(Gen(\Gamma, \varphi)^\circ)$ is a conjunction of properties of the form $\tilde{S}(S(\varphi^\circ))$ where $\tilde{S}$ ranges over all ground substitutions with $Dom(\tilde{S}) = FV(\varphi) \setminus FV(\Gamma)$ so by the rule **Conj** this amounts to showing

$$S(\Gamma^\circ) \cup \{f : S(Gen(\Gamma, \varphi)^\circ)\} \vdash_\wedge e : \tilde{S}(S(\varphi^\circ))$$

for all $\tilde{S}$. Such an $\tilde{S}$ does not affect the assumptions in the above judgment since all their free property variables are in $FV(\Gamma)$ so we can rewrite the judgment to

$$\tilde{S}(S(\Gamma^\circ)) \cup \{f : \tilde{S}(S(Gen(\Gamma, \varphi)^\circ))\} \vdash_\wedge e : \tilde{S}(S(\varphi^\circ)).$$

Applying Lemma 3.4 on the second part of the hypothesis we have

$$\tilde{S}(S(Gen(\Gamma, \varphi)^\circ)) \leq \tilde{S}(S(\bigwedge_{i\in I} \varphi_i^\circ))$$

and now the desired judgment follows from the one given by the induction hypothesis by using the rule **Weak** to strengthen the assumption on f in $(*)$. □

5 An inference algorithm

The polymorphic strictness logic defined in Figure 4 is syntax-directed; we now exploit this fact to define an inference algorithm for the logic. The inference algorithm $\mathcal{P}$, shown in Figure 7, accepts as argument a term e to be analysed and produces a triple consisting of a property φ, a set Γ of assumptions on the free variables of e and a constraint set C of equalities and inequalities between properties. The constraints in C limit the way in which the free variables in φ and Γ can be instantiated.

This way of defining type inference algorithm has previously been proposed by Damas (algorithm T in [8]) and has been applied to inference of intersection types for pure lambda terms by Coppo and Giannini [7] and for lambda terms extended with recursion by Jim [20]. It differs from algorithm W of Damas and Milner [9] by at the same time deducing a property of a term and assumptions on its free variables instead of taking as input a set of properties of the free variables.

General intersection type inference is known to be undecidable. Even decidable restrictions of the system possesses a feature that makes inference complicated. This is illustrated by the term $\lambda f.(f\ e_2)$ where only future bindings of f will determine how many properties and therefore how many sub-deductions are needed for e_2. This *a posteriori* expansion of proofs requires a complex operation for copying sub-deductions when new properties of e_2 are needed (see *e.g.* [6] or [29]). This expansion operation is too costly for our purposes and we opt for the alternative which is to stipulate that if the number of sub-deductions needed cannot be determined when an expression is analysed then it is fixed to 1. Any deduction needing more than one property of an occurrence of a sub-expression will be coerced into a deduction where the same property is used for that occurrence. The technical device to achieve this is called rank-2 intersection types, a notion that we now generalise to our strictness types. We emphasise that this does not go against polyvariance of the analysis since different *occurrences* of a variable still can have different properties assigned.

Rank 0:

$$\mathtt{t}^\sigma \in L^0(\sigma) \qquad \alpha^\sigma \in L^0(\sigma)$$

$$\frac{\varphi \in L^0(\sigma) \quad \alpha_i \in L^0(\flat) \quad i \in I}{\varphi ?\bigwedge_{i\in I} \alpha_i \in L^0(\sigma) \quad \varphi \Leftarrow \bigwedge_{i\in I} \alpha_i \in L^0(\sigma)}$$

$$\frac{\varphi_1 \in L^0(\sigma) \quad \varphi_2 \in L^0(\tau)}{\varphi_1 \to \varphi_2 \in L^0(\sigma \to \tau)}$$

Rank 1:

$$\frac{\varphi_i \in L^0(\sigma) \quad i \in I}{\bigwedge_{i\in I} \varphi_i \in L^1(\sigma)}$$

Rank 2:

$$\frac{\varphi_1 \in L^0(\flat) \quad \varphi_2 \in L^0(\flat)}{\varphi_1 \wedge \varphi_2 \in L^0(\flat)} \qquad \frac{\varphi \in L^0(\sigma)}{\varphi \in L^2(\sigma)}$$

$$\frac{\varphi_1 \in L^1(\sigma) \quad \varphi_2 \in L^2(\tau)}{\varphi_1 \to \varphi_2 \in L^2(\sigma \to \tau)}$$

Figure 6: Ranked strictness properties.

5.1 Ranked strictness properties

The algorithm and the reduction of inequalities rely on the properties involved being of a certain shape. We formalise this notion of shape by extending the notion of ranked types to our strictness properties. Rank-2 intersection types [23, 20] do not allow $\wedge$ at top-level or at the left of more than one $\to$. Our L^2-types are slightly more general in that they allow conjunctions *of base type* at top level. A feature of rank-2 type inference is that type inequalities of the form $\varphi_1 \wedge \varphi_2 \leq \varphi$ at higher types are avoided; with our L^2-types we still obtain this. For each type σ we define the three subsets $L^0(\sigma), L^1(\sigma), L^2(\sigma)$ by the rules in Figure 6. $L^0(\sigma)$ only allows conjunctions to appear to the right of the ?- and $\Leftarrow$-constructors in conditional properties and $L^1(\sigma)$ contains all top-level conjunctions of $L^0(\sigma)$-properties.

We say that an inequality $\varphi \leq \psi$ is a (p,q)-inequality if $\varphi \in L^p(\sigma)$ and $\psi \in L^q(\sigma)$. During inference, we have to reduce inequalities of the form $\forall\overline{\alpha}.\varphi \leq \psi$ where $\varphi \in L^2(\sigma)$ and $\psi \in L^1(\sigma)$. Since $\psi \in L^1(\sigma)$ it has form $\psi_1 \wedge \ldots \wedge \psi_n$ where $\psi_i \in L^0(\sigma)$ so the above inequality reduces to a set of inequalities $\{\forall\overline{\alpha}.\varphi \leq \psi_i : i \in I\}$. This can be turned into a set of $(2,0)$-inequalities by instantiating the $\overline{\alpha}$ with fresh variables, yielding the set $\{\varphi[\overline{\beta_i}/\overline{\alpha}] \leq \psi_i : i \in I\}$.

The inference algorithm is shown in Figure 7. In the case of application of a function satisfying $(\bigwedge \varphi_i) \to \varphi$ we must take into account that each φ_i requires a deduction showing that the argument satisfies this property. Since $\mathcal{P}$ pro-

The variables β, β_1, β_2 used in the algorithm are required to be fresh variables.

$$\mathcal{P}(x^\sigma) \quad = \quad (\{x^\sigma : \beta^\sigma\}, \beta^\sigma, \emptyset)$$

$$\begin{array}{lll}
\mathcal{P}(\lambda x^\sigma.e) & = & \textbf{let } (\Gamma, \varphi, C) = \mathcal{P}(e) \textbf{ in} \\
& & \quad (\Gamma', \psi \rightarrow \varphi, C) \quad \textbf{if } \Gamma = \Gamma' \cup \{x^\sigma : \psi\} \\
& & \quad (\Gamma, \mathtt{t}^\sigma \rightarrow \varphi, C) \quad \textbf{if } x^\sigma \notin Dom(\mathit{\Gamma})
\end{array}$$

$$\mathcal{P}(c^\flat) \quad = \quad (\emptyset, \mathtt{t}^\flat, \emptyset)$$

$$\begin{array}{llll}
\mathcal{P}(e_1 e_2) & = & \textbf{let } (\Gamma_i, \varphi_i, C_i) = \mathcal{P}(e_i) & i = 1, 2 \\
& & \;\textbf{in case } \varphi_1 \textbf{ of} & \\
& & \quad \beta ? \varphi_b : \; \textbf{let } \Gamma_1^{'} = \Gamma_1[\beta_1 \rightarrow \beta_2 / \beta] & \\
& & \qquad\qquad\qquad\; C_1^{'} = C_1[\beta_1 \rightarrow \beta_2 / \beta] & \\
& & \qquad\qquad \textbf{in } (\Gamma_1^{'} + \Gamma_2, \beta_2 ? \varphi_b, \{\varphi_2 \leq (\beta_1 \Leftarrow \varphi_b)\} \cup C_1^{'} \cup C_2) & \\
& & \quad \bigwedge_{i \in I} \varphi_{1i} \rightarrow \varphi_{12} : \; \textbf{let } (\tilde{\Gamma}_i, \tilde{\varphi}_i, \tilde{C}_i) = \mathit{Rename}(\mathit{\Gamma_2}, \varphi_2, C_2) & \\
& & \qquad\qquad\qquad\qquad\quad (S, C) = \mathit{Reduce}(C_1 \cup \bigcup_{i \in I} \tilde{C}_i \cup \{\tilde{\varphi}_i \leq \varphi_{1i}\}) & \\
& & \qquad\qquad\qquad\quad \textbf{in } (S(\Gamma_1 + \sum_{i \in I} \tilde{\Gamma}_i), S(\varphi_{12}), C) &
\end{array}$$

$$\begin{array}{llll}
\mathcal{P}(e_1 \; op \; e_2) & = & \textbf{let } (\Gamma_i, \varphi_i, C_i) = \mathcal{P}(e_i) \textbf{ in} & op = \; +, =, \ldots \\
& & \quad (\Gamma_1 + \Gamma_2, (\varphi_1 \wedge \varphi_2), C_1 \cup C_2) & i = 1, 2
\end{array}$$

$$\mathcal{P}(\mathsf{fix}(\lambda f.e)) \quad = \quad \mathcal{P}(e) \qquad \textit{if } f \notin FV(e)$$

$$\begin{array}{llll}
\mathcal{P}(\mathsf{fix}(\lambda f.e)) & = & \textbf{let } (\Gamma \cup \{f : \varphi_1\}, \varphi_2, C) = \mathcal{P}(e) & \\
& & \qquad (S, C') = \mathit{Reduce}(\{\mathit{Gen}(\mathit{\Gamma}, \varphi_2) \leq \varphi_1\} \cup C) & \\
& & \;\textbf{in } (S(\Gamma), S(\varphi_2), C') & \textit{if } f \in FV(e)
\end{array}$$

$$\begin{array}{llll}
\mathcal{P}(\mathsf{if}\; e_1 \;\mathsf{then}\; e_2 \;\mathsf{else}\; e_3) & = & \textbf{let } (\Gamma_i, \varphi_i, C_i) = \mathcal{P}(e_i) & i = 1, 2, 3 \\
& & \qquad (S, C) = \mathit{Reduce}(\{\varphi_2 \leq \beta, \varphi_3 \leq \beta\} \cup C_1 \cup C_2 \cup C_3) & \\
& & \;\textbf{in } (S(\Gamma_1 + \Gamma_2 + \Gamma_3), S(\beta ? \varphi_1), C) &
\end{array}$$

Figure 7: The inference algorithm $\mathcal{P}$

duces a (assignment, property, constraint)-triple from which all valid deductions can be obtained, it suffices to use the function *Rename* for generating copies of the (assignment, property, constraint)-triple deduced for the argument with all free variables renamed using fresh variables. Each conjunct in the function type corresponding to an occurrence of the formal parameter is then matched separately against a fresh copy of the property deduced of the argument.

The following lemma implies that the **case** in the rule for application is exhaustive. It is proved by inspection of the algorithm.

Lemma 5.1 *For all* $(\Gamma, \varphi, C) = \mathcal{P}(e^{\tau_1 \to \tau_2})$ *we have*

- $\forall x.\Gamma(x) \in L^1$
- $\varphi \in L^2$
- C *consists of* $(2,1)$*-inequalities*

Furthermore, if $\varphi \in L^2(\tau_1 \to \tau_2)$ *then it is either of the form* $\beta^{\tau_1 \to \tau_2} ? \varphi_b$ *or of the form* $\varphi_1 \to \varphi_2$.

The inequalities between properties that appear in the inference rules are collected in the set C during inference. These inequalities are simplified using the function *Reduce* specified by the rewrite rules in Section 5.3, resulting in a substitution S and a set of remaining, unresolved inequalities B that at the end of the inference describe all valid instances of the deduced property.

5.2 Completeness

The inference algorithm is complete for a restricted class of deductions *viz.*, those in which each occurrence of a variable x^σ is assigned a property from $L^0(\sigma)$. Formally, this restriction amounts to adding a side condition to the axiom for variables in Figure 4 which then reads

$$\{x^\sigma : \varphi\} \vdash x^\sigma : \varphi \quad \varphi \in L^0(\sigma). \qquad (\mathbf{Var - L^0})$$

In order to state the completeness result we define that substitution S *satisfies* a set of constraints C, written $S \models C$ if it yields a set of valid inequalities when applied to each constraint in C. The algorithm $\mathcal{P}$ is parameterised on the function *Reduce* that reduces a set of constraints C to a substitution T and a residual set of constraints C'. The soundness of $\mathcal{P}$ requires that the reduced constraint set C' does not allow any new solutions *i.e.*, that for any substitution S'

$$S' \models C' \quad \Rightarrow \quad S' \circ T \models C,$$

If furthermore all solutions S to C can be expressed as $S' \circ T$ where $S' \models C'$ then *Reduce* is said to be solution preserving. We prove the following completeness theorem for the case where *Reduce* is the trivial $\mathit{Reduce}(C) = (\mathit{Id}, C)$ that does not perform any reductions, and conjecture that the theorem still holds if *Reduce* is solution preserving.

Theorem 5.2 *Let* $\mathcal{P}(e) = (\Gamma, \varphi, C)$. *If the proof of the judgment* $\Delta \vdash e : \psi$ *only uses the rule* $\mathbf{Var - L^0}$ *for variables then there exists a substitution* S *such that*

1. $S(\beta^\sigma) \in L^0(\sigma)$ *for all* $\beta^\sigma \in \mathit{Dom}(S)$
2. $\mathcal{S} \models C$
3. $\Delta = S(\Gamma)$
4. $S(\varphi) = \psi$

Proof: By induction over the structure of e. The cases for application and fixed points are given in Appendix A.

5.3 Simplification of constraints

Recall that all properties (including variables) are typed and all constraints are between properties of same type. In order to determine the possible solutions of a set of constraints we show how to reduce such a set to a collection of sets only containing base type inequalities. We do this by specifying two rewrite relations on sets of constraints. First, there are constraints that reduce in one way only:

$$\begin{array}{lll}
\{\mathtt{t} \le \varphi\} \cup C & \leadsto_1 & \{\mathtt{t} = \varphi\} \cup C \\
\{\varphi \le \mathtt{t}\} \cup C & \leadsto_1 & C \\
\{\alpha \le \varphi_1 \to \varphi_2\} \cup C & \leadsto_1 & \{\alpha_2 \le \varphi_2, \varphi_1 \le \alpha_1\} \cup C \\
 & & \cup \{\alpha_1 \to \alpha_2 = \alpha\} \\
\{\varphi_1 \to \varphi_2 \le \alpha\} \cup C & \leadsto_1 & \{\alpha_1 \le \varphi_1, \varphi_2 \le \alpha_2\} \cup C \\
 & & \cup \{\alpha_1 \to \alpha_2 = \alpha\} \\
\{\varphi^\flat \le \bigwedge_{i \in I} \varphi_i^\flat\} \cup C & \leadsto_1 & \{\varphi^\flat \le \varphi_i^\flat\}_{i \in I} \cup C
\end{array}$$

Each equality that arises is solved using unification (possibly after expanding $\mathtt{t}^{\tau_1 \to \tau_2}$ into $\mathtt{t}^{\tau_1} \to \mathtt{t}^{\tau_2}$) producing a substitution and a new set of inequalities. A *Reduce*-function can be obtained by repeatedly applying this procedure to a set of constraints. When no more rules can be applied and all equalities have been unified, the accumulated substitution and the reduced set of inequalities are returned by the function *Reduce*. Observe that all reductions are solution preserving and hence so is *Reduce*.

From the definition of the $\le$-relation in Section 3 it is clear, however, that some inequalities $\varphi_1 \le \varphi_2$ can be proved in more than one way as is *e.g.*, the case where the properties are function properties or conditional properties. In the following we specify a further set of reduction rules: each reduction now produces two sets of inequalities where each set represents one possible way of proving the equality. For example, if we want to prove $\varphi_1 ? \varphi_2 \le \psi$ we can either show $\varphi_1 \le \psi$ or $\varphi_2 = \mathtt{f}^\flat$. This gives rise to a second set of rewrite rules

$$\begin{array}{lll}
\{\varphi_1 ? \varphi_2 \le \varphi\} \cup C & \leadsto_2 & \{\varphi_1 \le \varphi\} \cup C, \\
 & & \{\varphi_2 = \mathtt{f}^\flat\} \cup C \\
\{\varphi \le \varphi_1 ? \varphi_2\} \cup C & \leadsto_2 & \{\varphi \le \varphi_1, \mathtt{t} = \varphi_2\} \cup C \\
 & & \{\varphi = \mathtt{f}, \varphi_2 = \mathtt{f}\} \cup C \\
\{\varphi \le \varphi_1 \Leftarrow \varphi_2\} \cup C & \leadsto_2 & \{\varphi \le \varphi_1\} \cup C, \\
 & & \{\varphi_2 = \mathtt{f}^\flat\} \cup C \\
\{\varphi_1 \Leftarrow \varphi_2 \le \varphi\} \cup C & \leadsto_2 & \{\varphi_1 \le \varphi, \varphi_2 = \mathtt{t}^\flat\} \cup C, \\
 & & \{\varphi = \mathtt{t}, \varphi_2 = \mathtt{f}^\flat\} \cup C \\
\{\varphi_1 \to \varphi_2 \le \psi_1 \to \psi_2\} \cup C & \leadsto_2 & \{\psi_1 \le \varphi_1, \varphi_2 \le \psi_2\} \cup C, \\
 & & \{\psi_2 = \mathtt{t}\} \cup C
\end{array}$$

The $(2,1)$-inequality generated when matching two $\to$ properties is reduced to a set of $(2,0)$-inequalities by matching against each conjunct in the same way as conjunctions are dealt with in the first set of rewrite rules.

If the initial set of constraints only contains $(2,1)$-inequalities then so will all the sets generated during reduction. A set of $(2,1)$-inequalities that cannot be further reduced consists of base type inequalities of the form $\bigwedge_{i \in I} \alpha_i^\flat \le \alpha^\flat$. Such sets can be solved in linear time using a standard work-set algorithm (see [27] for a recent account).

In order to estimate the number of constraints generated when analysing expression e using algorithm $\mathcal{P}$ we observe that every syntactic construct except application adds at most two constraints to the set C. Application such as $e = (\lambda f.e_1)e_2$ will copy the constraints generated by e_2 for every occurrence of f in e_1. The resulting set will in turn be copied if e is passed as argument in another application. This behaviour is observed in the expression

$$(\lambda f_1.e_1)((\lambda f_2.e_2)(\ldots(\lambda f_n.e_n)(e)\ldots))$$

if each $\lambda f_i.e_i$ is of type $(\tau \to \tau) \to (\tau \to \tau)$ and each f_i occurs at least twice in e_i.

The simplification of constraints is also a source of complexity. The copying done by $\leadsto_2$-reduction can result in an exponential number of constraints. It is therefore important to notice that it is safe to discard one or more sets of inequalities since this just means that certain solutions are ignored. In particular, a polynomial bound is obtained if we for any $\leadsto_2$-reduction discard one of the sets generated. The choice can either be done statically before reduction starts (as in [1]) or heuristics can used to guide the choice dynamically. This provides a means of trading computational cost for precision.

6 Related work

Aiken, Wimmers and Lakshman [2] define a system for typing and inserting dynamic type checks in functional programs. Their language of types is very expressive: it includes intersections, unions, polymorphism and ?-conditional types. The type inferred of a program has the form

$$\forall\overline{\alpha}.\tau \textit{ where } C.$$

Here C is a set of constraints that may restrict the instantiation of the type variables $\overline{\alpha}$. Their conditional types only involve the ?-conditional types. We have added the $\Leftarrow$-constructor to the language of properties; this allows us to rewrite formulae such that the ?- and the $\Leftarrow$-constructors only are applied to formulae of base types. The ?-types are used for a slightly different purpose in [2] *viz.*, to propagate information about which branch in a case- or if-statement was taken. The example given is $\lambda y.\mathsf{case}\ \mathsf{y}\ \mathsf{of}\ \textit{true}, \textit{zero} : \textit{false}, \textit{succ}(\textit{zero})$ which receives the type

$$\begin{aligned}&\forall\alpha.\alpha \to (\textit{zero}?(\alpha \cap \textit{true})) \cup (\textit{succ}(\textit{zero})?(\alpha \cap \textit{false}))\\ &\quad \textit{where } \{\alpha \subseteq \textit{true} \cup \textit{false}\}.\end{aligned}$$

When α later is instantiated to *true*, say, the type $\textit{true}\cap\textit{false}$ reduces to the undefined and the whole type expression reduces to $\textit{true} \to \textit{zero}$. There is no recursion operator in the language, all recursion is expressed using a coding of the Y combinator. For this reason there is no explicit rule for inferring properties of recursive definitions so it is not clear to what degree the analysis exploits some form of polymorphic recursion. The constraints arising during inference are solved using an extension of the technique of Aiken and Wimmers [1]. This technique handles more general types (*e.g.* union types) but prohibits a number of interesting properties and among those the fundamental property of bi-strictness $\mathtt{f} \to \mathtt{t} \to \mathtt{f} \wedge \mathtt{t} \to \mathtt{f} \to \mathtt{f}$.

Jim [20] advocates the use of rank-2 intersection types for typing the lambda calculus with a recursion operator, building on earlier work on ranked type systems and intersection types by among others Coppo, Giannini, Leivant and van Bakel [7, 23, 30]. Rank-2 intersection types only allow conjunctions to appear to the left of a single arrow in a functional type (*e.g.* $((\sigma_1 \wedge \sigma_2) \to \tau) \to \rho$ is rejected). The restriction to rank 2 makes the system decidable (which is not the case with general intersection types) while retaining the property of principal typing: given a typable term e there exists a pair (A, τ) such that $A \vdash e : \tau$ is derivable and any other pair (A', τ') constituting a typing of e is an instance of (A, τ). We do not know whether a similar result can be proved for our polymorphic analysis from Figure 4. The language considered by Jim does not include arithmetic or logical operations and a main feature of his inference algorithm is that all inequalities can be reduced to equalities solvable by unification. In our case , this would *e.g.*, mean replacing $\alpha \leq \mathtt{t}$ with $\alpha = \mathtt{t}$ which is clearly discarding solutions. The extra features of our language and the more complex properties involved required extensions to the theory to deal with the conditional types and imply that not all inequalities are solvable by unification.

Banerjee [3] shows how to annotate rank-2 intersection types with control-flow information to obtain a type-based control-flow analysis for the pure lambda calculus. His approach to achieving modularity is similar to ours and that of Jim in deducing (environment,property)-pairs for terms and the analysis is polyvariant due to the use of intersection types. The algorithm proposed by Banerjee does type inference at the same time as flow analysis and for that reason requires a more complex constraint solving procedure than our "typed" constraints. Banerjee only considers pure lambda calculus and remarks that using polymorphic recursion to deal with fixed points leads to an undecidable system. In contrast, the assumption of dealing with well-typed terms allows us to handle a richer language including fixed points.

Kuo and Mishra [21] propose a type inference strictness analysis without conjunctions. This is extended to conjunctions by Benton [4] and Jensen [15] but none of them propose inference algorithms for the system. Hankin and Le Métayer [11, 12] present a framework for implementing conjunctive program analyses by deriving an abstract machine for proof search from the logics, following the work of Hannan and Miller [13]. This abstract machine will for a given term e *and* property φ search for a proof of $\vdash_\wedge e : \varphi$. It is thus a method for checking that a program has a particular property rather than inferring a property for it.

Monsuez [25] uses abstract interpretation of an operational semantics for a higher-order, untyped language to find sets of polymorphic strictness and totality types. Using sets of types adds top-level conjunction of types to the type language. The sets of types are organised into a lattice using a power-domain ordering on types ordered by instantiation and the set of types for a given program is found by iterating over this lattice. Widening operators based on unification are used to collapse a set of types down to a single type in order to ensure the termination of the analysis. Since the widenings remove conjunctions from the type language they restrict the properties derived. It remains to be shown how much of this loss could be avoided by working in a typed language.

Henglein and Mossin [14] and Dussart, Henglein and Mossin [10] present a polymorphic binding-time analysis for an extension of the simply-typed lambda calculus. They do not include conjunctions or conditional types but introduce instead "qualified types" of the form $b_1 \leq b_2 \Rightarrow b$ where $b_1 \leq b_2$ is a constraint that applies to b, much in the style of the types of Aiken *et al.* above. Judgments in their logic are of the form

$$C, A \vdash e : \forall\overline{\alpha}.b$$

with A a set of conjunctions and C a set of constraints that must be satisfied for the deduction to be valid. Polymorphic recursion is used to analyse fixed points and mutually recursive definitions are handled by applying an improvement of Mycroft's iterative calculation of fixed points in a lattice of type schemes. The absence of conjunctions seems to be of less importance in binding-time analysis than it would be for our strictness analysis since there are less "bi-static" functions than bi-strict ones. For example, $e_1 + e_2$ is undefined as soon as one of e_1 and e_2 is undefined whereas it is static only if both e_1 and e_2 are static.

7 Conclusions

We have presented a strictness analysis for a higher-order functional language that combines polymorphism with conjunctions and conditional strictness properties. To the best of our knowledge, it is the first time this combination is used for strictness analysis. It results in a compact representation of strictness properties and in an analysis that is well suited to deal with modular programs. The analysis was proved correct using a translation into a monomorphic strictness logic with conjunctions. We have presented a constraint-based algorithm for inferring polymorphic properties together with rules for reducing the constraints generated during inference and proved that the algorithm is complete for deductions where occurrences of variables only are assigned the simpler rank-0 properties. The reduction rules give rise to a number of algorithms for solving the constraint sets, making it possible to vary the complexity of the constraint simplification from polynomial to exponential in the size of the initial constraint set.

This work can be extended in several directions. We believe that it should be possible to extend the polymorphic binding-time analysis of Dussart *et al.* [10] with conjunctions to make it fit into our framework, providing an analysis for module-oriented partial evaluation. Banerjee's control-flow analysis [3] is based on the same notion of ranked program properties as our analysis. It is worth investigating whether an extension to a typed language like ours can lead to an analysis that can deal with conditionals and fixed points. Furthermore, the basic principles behind our analysis is not limited to the functional language paradigm. In the realm of reactive systems, preliminary investigations indicate that the same approach can be applied to obtain a modular clock analysis for the synchronous language Signal [22]. In particular, the notion of polymorphic clocks is well suited to model how the response frequency of a reactive system depends on its context [17].

Work remains to be done in order to adopt our analysis for program fragments to a real module languages. We envisage an extension along the following lines, stressing that what follows is only a sketch of a possible approach. For a module

$$\mathbf{module\ M\ \ in\{g_i\}_{i\in I}\ \ out\{h_j = e_j\}_{j\in J}}$$

that imports a set of functions g_i and exports the functions h_j we obtain the *inferred interface* of M by calculating $(\Gamma_j, \varphi_j, C_j) = \mathcal{P}(e_j)$ for all $j \in J$. The hypotheses on the imported functions made in each e_j are listed in Γ_j so the sum $\sum_{j\in J} \Gamma_j$ represents the conjunction of all hypotheses made in the module. Similarly, all constraints generated during analysis of the e_j's must be unioned. The inferred interface for M is then

$$(\Gamma_{in}, \Gamma_{out}, C_M) = (\sum_{j\in J} \Gamma_j, \{h_j : \varphi_j\}_{j\in J}, \bigcup_{j\in J} C_j)$$

where the imported and the exported functions are described by an environment associating names with properties together with a set of constraints.

Linking module M with interface as above with a module N where the imports of M are defined is done as follows. Assuming that N has interface $(\Delta_{in}, \Delta_{out}, C_N)$ we calculate

$$(S, C_r) = \mathit{Reduce}(\{\Delta_{out}(g_i) \leq \Gamma(g_i)\}_{i\in I})$$

deducing the constraints that must hold for the exported g_i to satisfy the hypotheses made on them in M. The resulting module $M(N)$ then has interface

$$(\{\}, \{h_j : S(\varphi_j)\}_{j\in J}, S(C_M) \cup S(C_N) \cup C_r).$$

A third issue to be addressed is how to extend the analysis to deal with algebraic and polymorphic datatypes. For polymorphism, it would be worth investigating for what kind of properties polymorphism in the type of an expression implies polymorphism in the strictness property of the expression. For modelling algebraic datatypes, we have proposed an approach based on uniform properties [16, 19] that only describe the content of a data structure. For lists with elements of type τ we have *e.g.* $\Box\varphi^\tau$, meaning that every element of a list satisfies φ, and $\Diamond\varphi^\tau$, meaning that at least one element of a list satisfies φ. For the polymorphic function `map` : $(\alpha \to \beta) \to \mathit{List}\ \alpha \to \mathit{List}\ \beta$ where α, β range over types we observe that we have the two properties $(\alpha \to \beta) \to \Box\alpha \to \Box\beta$ and $(\alpha \to \beta) \to \Diamond\alpha \to \Diamond\beta$ obtained by replacing the type constructor with a property constructor. A pertinent question is to what extent this observation generalises to other operators and data types.

Acknowledgments Thanks are due to Anindya Banerjee, Radhia Cousot, Andrew Kennedy, Daniel Le Métayer and the referees for discussions related to and/or comments on an earlier version of this paper.

References

[1] A. Aiken and E. Wimmers. Type inclusion constraints and type inference. In *Proc. of Functional Programming Languages and Computer Architecture (FPCA'93)*, pages 31–41. ACM Press, 1993.

[2] A. Aiken, E. L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *Proc. of the 21st Annual ACM Symposium on Principles of Programming Languages*, pages 163–173. ACM Press, 1994.

[3] A. Banerjee. A modular, polyvariant and type-based closure analysis. In *Proc. ACM Int. Conf. on Functional Programming (ICFP'97)*, pages 1–10. ACM Press, 1997.

[4] P. N. Benton. Strictness logic and polymorphic invariance. In A. Nerode and M. Taitslin, editors, *Proc. of the 2. International Symposium on Logical Foundations of Computer Science, Tver, Russia*, LNCS vol. 620. Springer, 1992.

[5] G. L. Burn, C. Hankin, and S. Abramsky. The theory and practice of strictness analysis for higher order functions. *Science of Computer Programming*, 7:249–278, 1986.

[6] M. Coppo, M. Dezani–Ciancaglini, and B. Venneri. Principal type schemes and lambda calculus semantics. In J.P. Seldin and J.R. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 535–560. Academic Press, 1980.

[7] M. Coppo and P. Giannini. A complete type inference algorithm for simple intersection types. In Jean-Claude Raoult, editor, *Proc. of CAAP '92, 17th Colloquium on Trees in Algebra and Programming*, LNCS vol. 581, pages 102–123. Springer-Verlag, 1992.

[8] L. Damas. *Type assignment in programming languages.* PhD thesis, Univ. of Edinburgh, 1985.

[9] L. Damas and R. Milner. Principal type schemes for functional programs. In *Proc. of 9th ACM Symposium on Principles of Programming Languages*, pages 207–212. ACM Press, New York, 1982.

[10] D. Dussart, F. Henglein, and C. Mossin. Polymorphic recursion and subtype qualifications: Polymorphic binding-time analysis in polynomial time. In *Proc. 2nd Int. Static Analysis Symposium (SAS), Glasgow, Scotland*, LNCS vol. 983. Springer, 1995.

[11] C. Hankin and D. Le Métayer. Deriving algorithms from type inference systems: Applications to strictness analysis. In *Proc. of 21st ACM Symp. on Principles of Programming Languages*, pages 202–212. ACM Press, 1994.

[12] C. Hankin and D. Le Métayer. Lazy type inference and program analysis. *Science of Computer Programming*, 25:219–249, 1995.

[13] J. Hannan and D. Miller. From operational semantics to abstract machines. *Mathematical Structures in Computer Science*, 2(4), 1992.

[14] F. Henglein and C. Mossin. Polymorphic binding-time analysis. In D. Sannella, editor, *Proc. of 5th European Symposium on Programming*, LNCS vol. 788, pages 287–301. Springer Verlag, 1994.

[15] T. Jensen. Strictness analysis in logical form. In J. Hughes, editor, *Proc. of 5th ACM Conference on Functional Programming Languages and Computer Architecture*, LNCS vol. 523, pages 352–366. Springer, 1991.

[16] T. Jensen. Abstract interpretation over algebraic data types. In H. Bal, editor, *Proc. 5th IEEE International Conference on Computer Languages.* IEEE Press, 1994.

[17] T. Jensen. Clock analysis of synchronous dataflow programs. In W. Scherlis, editor, *Proc. of ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 156–167, San Diego, 1995. ACM Press.

[18] T. Jensen. Conjunctive type systems and abstract interpretation of higher-order functional programs. *Journal of Logic and Computation*, 5(4):397–421, 1995.

[19] T. Jensen. Disjunctive program analysis for algebraic data types. *ACM Transactions on Programming Languages and Systems*, 19(5):752–804, 1997.

[20] T. Jim. What are principal typings and what are they good for? In *Proc. of 23rd Symposium on Principles of Programming Languages*, pages 42–54. ACM Press, 1996.

[21] T-M Kuo and P. Mishra. Strictness analysis : A new perspective based on type inference. In *Proc. 4th. Int. Conf. on Functional Programming and Computer Architecture.* ACM Press, 1989.

[22] P. Le Guernic, T. Gauthier, M. Le Borgne, and C. Le Maire. Programming real-time applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1321–1336, 1991.

[23] D. Leivant. Polymorphic type inference. In *Proc. of the Tenth Annual ACM Symposium on Principles of Programming Languages*, pages 88–98. ACM Press, 1983.

[24] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[25] B. Monsuez. Using abstract interpretation to define a strictness type inference system. In W. Scherlis, editor, *Proc. of ACM Symposium on Partial Evaluation and Semantics-based program manipulation (PEPM'95)*, pages 122–133. ACM Press, 1995.

[26] A. Mycroft. Polymorphic type schemes and recursive definitions. In *Proc. of Int. Symposium on Programming, Toulouse, France*, pages 217–239. Springer LNCS vol. 167, 1984.

[27] J. Rehof and T. Mogensen. Tractable constraints in finite semilattices. In R. Cousot and D. Schmidt, editors, *Proc. of 3rd Int. Static Analysis Symposium (SAS'96)*, pages 285–300. Springer LNCS vol. 1145, 1996.

[28] J. C. Reynolds. Automatic computation of data set definitions. *Information Processing*, 68, 1969.

[29] S. Ronchi Della Rocca. Principal type scheme and unification for intersection type discipline. *Theoretical Computer Science*, 59, 1988.

[30] S. v. Bakel. Complete restrictions of the intersection type discipline. *Theoretical Computer Science*, 102(1):135–163, 1992.

A Proof of Theorem 5.2

Theorem: Let $\mathcal{P}(e) = (\Gamma, \varphi, C)$. If the proof of the judgment $\Delta \vdash e : \psi$ only uses the rule $\mathbf{Var - L^0}$ for variables then there exists a substitution S such that

1. $S(\beta^\sigma) \in L^0(\sigma)$ for all $\beta^\sigma \in Dom(S)$
2. $\mathcal{S} \models C$
3. $\Delta = S(\Gamma)$
4. $S(\varphi) = \psi$

Proof: By induction over the structure of e. We give the cases for application and fixed points.

Application: Assume that $\Delta \vdash_\wedge e_1 e_2 : \varphi?\varphi_b$ has been deduced using the rule **App** from the premises

- $\Delta_1 \vdash e_1 : ((\bigwedge_{i\in I} \varphi_i) \to \varphi)?\varphi_b$
- $\Delta_i \vdash e_2 : \psi_i$
- $\forall i \in I : \psi_i \leq \varphi_i$

Then $\Delta = \Delta_1 + \sum_{i\in I} \Delta_i$. Let $(\Gamma_1, \varphi_1, C_1) = \mathcal{P}(e_1)$ and similar for e_2. By induction hypothesis there exists S_1 such that

- $S_1 \models C_1$
- $\Delta_1 = S_1(\Gamma_1)$
- $S_1(\varphi_1) = ((\bigwedge_{i\in I} \varphi_i) \to \varphi)?\varphi_b$

and S_i satisfying

- $S_i \models C_2$
- $S_i(\Gamma_2) = \Delta_i$
- $S_i(\varphi_2) = \psi_i$

We recall that there are no common variables between the sub-deductions hence substitutions $S_1, \ldots, S_i, \ldots$ have disjoint domains.

For $\varphi_1 \in L^2(\tau_1 \to \tau_2)$ there are two possibilities:

$$\varphi_1 = \beta^{\tau_1 \to \tau_2}?\varphi' \qquad \text{and} \qquad \varphi_1 = (\bigwedge_{\mathrm{i}\in \mathrm{I}} \varphi_{1\mathrm{i}}) \to \varphi_{12}.$$

In the first case we have that

$$S_1(\beta?\varphi') = S_1(\beta)?S_1(\varphi') = ((\bigwedge_{i\in I} \varphi_i) \to \varphi)?\varphi_b$$

so

$$S_1(\beta) = (\bigwedge_{i\in I} \varphi_i) \to \varphi \in L^0(\tau_1 \to \tau_2)$$

which implies that I is singleton. Hence the substitution

$$S = S_1[\beta_1 \mapsto \varphi_i, \beta_2 \mapsto \varphi] \circ S_i$$

where β_1 and β_2 are the new variables introduced by the algorithm satisfies

$$S(\varphi_2) = S_i(\varphi_2) = \psi_i \leq \varphi_i = S(\beta_1) \leq S(\beta_1 \Leftarrow \varphi').$$

Furthermore, $S(\beta_2?\varphi_b) = \varphi?\varphi_b$.

In the second case we have

$$S_1((\bigwedge_{i\in I} \varphi_{1i}) \to \varphi_{12}) = \bigwedge_{i\in I} S(\varphi_{1i}) \to S(\varphi_{12}) = \bigwedge_{i\in I} \varphi_i \to \varphi$$

so $S_1(\varphi_{1i}) = \varphi_i$ and $S(\varphi_{12}) = \varphi$.

If $(\tilde{\Gamma}_i, \tilde{\varphi}_i, \tilde{C}_i) = \mathit{Rename}(\Gamma_2, \varphi_2, C_2)$ then let S_i be the substitution obtained from S_2 by prefixing it with the same renaming. Then the substitution

$$S = S_1 \circ \bigcirc_{i\in I} S_i$$

satisfies

$$S(\tilde{\varphi}_i) = S_i(\varphi_2) = \psi_i \leq \varphi_i = S_1(\varphi_{1i}) = S(\varphi_{1i}).$$

The other inequalities are still satisfied since from $\mathcal{S}_1 \models C_1$, $\mathcal{S}_{2i} \models C_{2i}$ and the disjointness of the substitutions it follows that $\mathcal{S} \models C_1 \cup \bigcup_{i\in I} C_{2i}$, Finally, S satisfies

$$S_1 \circ \bigcirc_{i\in I} S_{2i}(\varphi_{12}) = S_1(\varphi_{12}) \leq \psi.$$

It is straightforward to verify that the resulting environment satisfies 2. in either case.

Fixed points. Assume that $f \in FV(e)$ and that

$$\mathcal{P}(e) = (\Delta \cup \{f : \varphi_1\}, \varphi_2, C).$$

Assume that $\Gamma \vdash \mathsf{fix}\ \lambda f.e : \varphi$ has been deduced from

$$\Gamma \cup \{f : \bigwedge_{i\in I} \varphi_i\} \vdash e : \varphi \quad \text{and} \quad \forall \mathrm{i} \in \mathrm{I} : \mathit{Gen}(\Gamma, \varphi) \leq \varphi_i.$$

By induction hypothesis there is a substitution S such that

- $S(\varphi_1) = \bigwedge_{i\in I} \varphi_i$
- $S(\varphi_2) = \varphi$
- $S(\Delta) = \Gamma$

We only need to check that the added constraint

$$\mathit{Gen}(\Gamma, \varphi_2) \leq \varphi_1$$

is satisfied by S. For this, we first observe that

$$FV(\varphi) \cap FV(\varphi_2) = \emptyset$$

and

$$FV(\mathit{Gen}(S(\Delta), S(\varphi_2))) \subseteq FV(S(\varphi_2)).$$

This implies that

$$FV(\mathit{Gen}(S(\Delta), S(\varphi_2))) \cap FV(\varphi_2) = \emptyset.$$

Since $\mathit{Gen}(\Delta, \varphi_2) \leq S(\varphi_2)$ we then have that

$$S(\mathit{Gen}(\Delta, \varphi_2)) \leq S(\mathit{Gen}(S(\Delta), S(\varphi_2)).$$

The satisfaction of the extra constraint now follows from

$$S(\mathit{Gen}(\Delta, \varphi_2)) \leq S(\mathit{Gen}(S(\Delta), S(\varphi_2)) = S(\mathit{Gen}(\Gamma, \varphi)) \\ \leq S(\bigwedge_{i\in I} \varphi_i) = S(S((\varphi_1)) = S((\varphi_1)).$$

$\square$