Under consideration for publication in Math. Struct. in Comp. Science

Modular Control-Flow Analysis with Rank 2 Intersection Types

Anindya Banerjee[1†] and Thomas Jensen[2‡]

[1] *Department of Computing and Information Sciences*
Kansas State University
Manhattan KS 66506, U.S.A.
e-mail: ab@cis.ksu.edu
[2] *IRISA/CNRS*
Campus de Beaulieu
F-35042 Rennes, France
e-mail:jensen@irisa.fr

Received 26 December 2001

We show how the principal typing property of the rank 2 intersection type system enables the specification of a modular and polyvariant control-flow analysis.

1. Introduction

The performance of optimising compilers crucially depends on the availability of control-flow information at compile time. For any first-order imperative program, such information is available via a flowchart constructed from the program text. Consequently, traditional dataflow analyses can be used to perform a series of compile-time program optimisations (Aho et al. 1986). For higher-order programs, however, a control-flow graph is often not evident from the program text. In these programs, even simple control structures like while loops are implemented using functions and computation is hidden under a single operation: function application. Hence, a number of *control-flow analyses* have been proposed (Jones 1981; Sestoft 1988; Shivers 1991; Jagannathan and Weeks 1995) for higher-order programs, all of which seek to answer the fundamental question: given a program point, what functions can be the result of evaluation at the program point? Given this information, the call-graph of a program can be constructed and a suite of compiler optimisations *e.g.*, closure conversion (Steckler and Wand 1997; Dimock et al. 2001), useless variable elimination (Wand and Siveroni 1999), constant propagation, induction variable elimination (Shivers 1991) *etc.* can be enabled.

[†] Partially supported by postdoctoral fellowships at the Laboratoire d'Informatique, École Polytechnique, Palaiseau, France (thanks to Radhia Cousot) and at DAIMI, University of Aarhus, Denmark (thanks to Flemming Nielson); and by NSF grant EIA-9806835.
[‡] Partially supported by the European IST Future and Emerging Technologies scheme (project IST-1999-29075 "Secsafe")

The classical technique for control-flow analysis is abstract interpretation (Cousot and Cousot 1977) of either the denotational semantics of the underlying language (Shivers 1991) or of its operational semantics (Nielson and Nielson 1997). An equivalent technique uses a system of constraints to specify control-flow analysis so that flow information is obtained as the minimal solution of the constraint system (Palsberg 1995). Both techniques require whole-program analysis. For control-flow analysis of program fragments containing free variables, the above techniques usually assume an environment that associates a property with each free variable or assume that only trivial properties hold of the free variables. Neither assumption is satisfactory: the former requires re-analysis whenever the environment changes; the latter leads to poor analysis results and the rejection of program fragments that require functions as inputs.

We describe an algorithm for modular and polyvariant control-flow analysis of simply-typed program fragments. The result of analysing a program fragment P, is a pair containing a property *and* an environment: the pair describes a *relation* between the properties of the free variables of P and the property of the entire program fragment P. The environment provides a summary of the minimum set of constraints that must be satisfied by any other program fragment that may link to P. Our algorithm computes a "principal solution": the environment and property of any other program fragment that links to P can be obtained as an instance of the environment and property computed for P (see Theorem 6.4 for the precise statement). Thus no re-analysis of P is required.

Recently, there has been much interest in using annotated type systems for program analysis. The intuition is that types and expressions can be annotated with the properties of interest, *e.g.*, control-flows (Tang 1994; Heintze 1995; Banerjee 1997), binding times (Nielson and Nielson 1992; Hatcliff and Danvy 1997; Nielson and Nielson 1998), strictness (Kuo and Mishra 1989; Jensen 1991; Benton 1992; Jensen 1998) effects (Talpin and Jouvelot 1994), regions (Tofte and Talpin 1994; Tofte and Talpin 1997), concurrent behaviours (Amtoft et al. 1997), dead-code (Damiani 1996; Damiani and Prost 1996; Coppo et al. 1998; Kobayashi 2000) *etc.*, so that if an expression, e, has the annotated type τ, then evaluation of the expression exhibits the properties described by the annotation of τ. Static analysis of the expression e is then synonymous with the calculation, via an annotated-type inference algorithm, of its property annotations. For control-flow analysis, we annotate every function in an expression with a label, and associate a set of function labels, φ, with every function type, τ. The intuition is that if e evaluates to a closure, then the text of the closure is a function whose label is in φ. The static determination of the set of function labels that e can possibly evaluate to is thus synonymous with the calculation, via an inference algorithm, of its flow annotations.

An advantage of the type-based approach is that it provides a method for performing *compositional* and *modular* program analysis. "Compositional" means that the analysis of an expression is derived through the composition of the analyses of its proper subparts. We say that an analysis is "modular" if it can analyse program fragments containing free variables in isolation, and if the linking of fragments does not require their re-analysis. A modular program analysis thus seems indispensable for *separate compilation*; however, the extent to which modular analyses can be used in practice for efficient separate compilation, remains to be seen. The effective use of the information obtained from a modular

analysis depends on link-time optimisations; but such optimisations are rarely done in industrial-strength compilers at the present point of time.

1.1. *Goals and methods*

The goal of this article is to provide an algorithm for modular control-flow analysis of simply-typed functional programs. For precision, we also demand that our analysis be *polyvariant*, *i.e.*, it must annotate a function with different properties at its different application sites.

How can we achieve a modular analysis? We make the following observation due to Damas (Damas 1985, Chapter 1): the simply-typed lambda calculus satisfies the *principal typing property*. This means, given a typable program fragment, e, possibly containing free variables, there is a pair, $\langle \Gamma, \tau \rangle$, such that $\Gamma \vdash e : \tau$ represents all valid typings (*i.e.*, type-derivation trees) of e. Furthermore, there is an algorithm that calculates such a pair for e. The significance is, first, the user does *not* have to supply the types of the free variables of e – they can be inferred automatically; this means, one can type all *uses* of a free variable independently of its *definitions* – a feature crucial for analysing program fragments. Second, when fragments are linked, the typing of e might possibly change – but principality guarantees that the new typing is always a substitution instance of $\Gamma \vdash e : \tau$. Thus we can avoid a re-inference of e upon linking. Principal typing is thus a simple way of achieving a modular analysis. In the rest of the paper, we apply this idea to obtain a modular closure analysis.

How can we achieve a polyvariant analysis? Since a function may be applied to different arguments and may return different results at each of its application sites, we can represent the different behaviours as an intersection type (Sallé 1978; Coppo et al. 1980a; Coppo et al. 1980b). This idea has been used in other contexts, *e.g.*, strictness analysis. In such analyses, the ability to form intersections of types have proven essential since they allow expressing several properties of a function in one formula. (Jensen 1991; Jensen 1995) proved that by adding intersections to a language of strictness types, we obtain an analysis in the style of intersection types which is equivalent in power to the abstract interpretation-based strictness analysis of (Burn et al. 1986).

The intersection types we will consider are the "rank 2 intersection types". Intuitively, the rank of a type describes the nesting of functions in argument position. The notion of rank has been used to identify restricted versions of intersection type systems for which type inference is decidable. In particular, type inference in the rank 2 intersection type system yields principal typings (Jim 1996). Thus, we can achieve our goal of providing a modular and polyvariant analysis. Further, the rank 2 intersection type system types significantly more terms than core ML, and moreover, can assign more general principal types to some core ML terms than the ML type system (van Bakel 1993). While recent results by Kfoury and Wells indicate the existence of principal typings and show decidability of type inference in systems of finite-rank intersection types (Kfoury and Wells 1999), we have not pursued this direction. Rather, we restrict ourselves to rank 2 intersection types because the constraints resulting from an analysis based on such types are simple to solve. In technical terms, we avoid having to solve constraints of the form

$\varphi_1 \wedge \varphi_2 \leq \psi$ over function domains; this is also the reason why we have not considered the full intersection type discipline.

The inference algorithm that implements the control-flow analysis deals with the subtyping that naturally arises from the fact that one program property implies other properties. We follow the standard approach to type inference in the presence of subtyping by letting the result of typing a term e be a *triple*, (Γ, φ, C), where φ is a property of e, Γ is a set of assumptions on the free variables of e and C is a set of constraints. Property variables are used to describe the dependencies between the property φ and the properties in the environment. The constraint set C limits the way in which the property variables in Γ and φ can be instantiated.

1.2. *A small typed functional language*

We choose a language with simple types as our base language. The language, called vPCF (Riecke 1991), is essentially call-by-value PCF (Plotkin 1977) with recursion, conditionals and basic arithmetic. The types and terms of the language are given by following grammar, where op abbreviates either the succ or the pred operations.

$$\begin{array}{rcl} \sigma, \tau & ::= & \mathsf{int} \mid \sigma \to \sigma \\ v & ::= & n \mid \lambda x^{\sigma}.\, e \mid \mathsf{fun}\, f^{\sigma \to \tau}\, (x^{\sigma}) = e \\ e & ::= & v \mid x \mid e_1 e_2 \mid \mathsf{op}\ e \mid \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \end{array}$$

Values, v, in vPCF are integers, lambda abstractions and recursive function definitions. Since we have anonymous functions in the language via lambda abstractions, we insist that in a recursive function declaration, $\mathsf{fun}\, f^{\sigma \to \tau}\, (x^{\sigma}) = e$, the function name, f, and the parameter, x, both appear in e. The type system for vPCF is the standard one and is omitted save the rule for recursion.

$$\frac{\Gamma \oplus \{f : \sigma \to \tau\} \oplus \{x : \sigma\} \vdash e : \tau}{\Gamma \vdash \mathsf{fun}\, f^{\sigma \to \tau}\, (x^{\sigma}) = e : \sigma \to \tau}$$

Γ denotes the type environment, in which identifiers are associated with simple types. The notation $\Gamma \oplus \{x : \sigma\}$ means "extend the environment Γ with the binding $\{x : \sigma\}$, where $x \notin \mathit{Dom}(\Gamma)$".

1.3. *Organisation*

The rest of the article is structured as follows. A brief review of control-flow analysis is given in Section 2 by means of an example. Section 3 introduces the language of properties, defines the rank 2 intersection properties and shows how to define a "generic" control-flow property for each type. Section 4 introduces the language ℓPCF and specifies, via inference rules, a property system for polyvariant control-flow analysis for ℓPCF. Orderings on ranked properties, needed in the inference rules, are introduced and several properties of the ranked types and of the inference rules are shown in this section.

Section 5 provides the small-step operational semantics and type soundness results for ℓPCF. Section 6 provides the type inference algorithm for inferring control-flow information, and proves soundness (Section 6.2) and completeness (Section 6.3) of the algorithm. The soundness of the inference algorithm, in conjunction with type soundness, shows correctness of the analysis: this is explained at the end of Section 6.2. Section 7 surveys related work and Section 8 concludes with a discussion.

This article grew out of two papers. (Banerjee 1997) developed a modular and polyvariant control-flow analysis for untyped programs that infers control flow information and types at the same time. (Jensen 1998) proposed a modular strictness analysis for typed programs based on intersection types and parametric polymorphism. Working with a typed language means that the analysis can use powerful induction principles (akin to polymorphic recursion) and still guarantee that the analysis terminates. In this article, we show how a control-flow analysis for a typed, higher-order functional language can be designed based on the techniques developed in these previous works.

2. Control-flow analysis

Consider an expression in vPCF such that all lambda abstractions and recursive function definitions in this expression are labelled uniquely. Let λ^{ℓ} and $\mathsf{fun}^{\ell'}$ refer to the lambda abstraction labelled ℓ and the recursive function labelled ℓ'. A node in the abstract syntax tree of an expression is called a "program point". Then, given a closed expression, control-flow analysis (CFA) seeks to answer the following question:

What set of function labels (that is, labelled lambda abstractions or recursive function definitions) can each program point possibly evaluate to?

In particular, if the program point is an application site, the question is the same as:

What set of function labels (that is, labelled lambda abstractions or recursive function definitions) can be called from the application site?

We give an example of CFA below. For exact details of the usual abstract interpretation-based analysis, we refer the reader to the works (Sestoft 1991) and (Shivers 1991) who provide two distinct control-flow analyses using abstract interpretation; see (Mossin 1997b) for a comparison of the two analyses.

Example: The term

$$T = (\lambda^1 g^{(\mathsf{int}\to\mathsf{int})\to(\mathsf{int}\to\mathsf{int})}.\ g\ (g\ (\lambda^2 v^{\mathsf{int}}.\ v)))\ \ (\lambda^3 x^{(\mathsf{int}\to\mathsf{int})}.\ \lambda^4 y^{\mathsf{int}}.\ y)$$

with type $\mathsf{int}\to\mathsf{int}$, will serve as a running example throughout the article. Control-flow analysis of T yields the following results:

1 The function part of the application, $\lambda^1 g.\ g(g(\lambda^2 v.\ v))$, yields $\{1\}$.
The variable g yields $\{3\}$.
$\lambda^2 v.\ v$ yields $\{2\}$.
The application, $g(\lambda^2 v.\ v)$, yields $\{4\}$.
The application, $g(g(\lambda^2 v.\ v))$, yields $\{4\}$.

2 The argument part of the application, $\lambda^3 x.\ \lambda^4 y.\ y$, yields $\{3\}$.
y yields $\varnothing$.
$\lambda^4 y.\ y$, yields $\{4\}$.
3 The entire expression T yields $\{4\}$.

The interesting case is that of the function, λ^3: it once gets applied to λ^2 and again to the result of this application, *i.e.*, to λ^4. Shivers's abstract interpretation-based 0CFA analysis would report that at each of its application sites, λ^3 is possibly applied to the set, $\{2, 4\}$. For a polyvariant analysis, however, we expect the analysis to report: at the application site $g(\lambda^2 v.\ v)$, λ^3 is applied to λ^2, and at the application site $g(g(\ldots))$, λ^3 is applied to λ^4. In the sequel, we will capture this polyvariance using intersection types.

3. A property system for polyvariant control-flow analysis

Here and in the following sections, we are motivated by the general framework for type inference in the presence of subtypes as defined by (Mitchell 1991) and extended to control-flow analysis by (Heintze 1995) and to behaviour analysis by (Amtoft et al. 1997; Amtoft et al. 1999). We show that the rank 2 fragment of the intersection type discipline can be instrumented to perform a polyvariant control-flow analysis. We first define the instrumented rank 2 intersection types (called rank 2 control-flow properties). The definition follows that of Jim (Jim 1995; Jim 1996), who builds on earlier works (Leivant 1983; van Bakel 1993; van Bakel 1996).

Let L be a countably infinite set of labels. For each type σ, we have an infinite set of *label variables* ranged over by ξ. The BNF of labels at a given type, σ, is specified below.

$$\mathit{Labels}(\sigma) \ni \kappa ::= \xi \mid L \mid \kappa_1 \cup \kappa_2.$$

For each type σ, define the *properties at* σ, $\mathit{Prop}(\sigma)$, to be the smallest set satisfying:

$$\mathsf{t}^{\mathsf{int}} \in \mathit{Prop}(\mathsf{int}) \qquad \frac{\varphi_i \in \mathit{Prop}(\sigma) \quad i \in I}{\bigwedge_{i \in I} \varphi_i \in \mathit{Prop}(\sigma)}$$

$$\frac{\varphi_1 \in \mathit{Prop}(\sigma) \quad \varphi_2 \in \mathit{Prop}(\tau) \quad \kappa \in \mathit{Labels}(\sigma \to \tau)}{(\varphi_1 \to \varphi_2, \kappa) \in \mathit{Prop}(\sigma \to \tau)}$$

For each type σ, we also have *ranked properties at* σ, denoted by $\mathit{Prop}_k(\sigma)$. A property has rank k, provided all intersection constructors are to the left of at most $k-1$ arrow constructors. A property at rank 0 has no intersection constructors. In this paper, we will only be interested in properties at ranks 0, 1, 2. Such properties are amenable to automatic inference and are defined in Table 1. Furthermore, we assume that the intersection operator, $\wedge$, is associative, commutative and idempotent. We can define a "generic" property for a given type σ, written σ^*, by decorating the type with fresh property variables. Formally, the translation $(.)^*$ from types to properties is defined by induction on the structure of types as shown below.

- $\mathsf{int}^* = \mathsf{t}^{\mathsf{int}}$
- $(\sigma \to \tau)^* = (\sigma^* \to \tau^*, \xi)$, ξ is fresh and $\xi \in \mathit{Labels}(\sigma \to \tau)$

Clearly, $\sigma^* \in \mathit{Prop}_0(\sigma)$, and labels, if any, in σ^* are all label variables. The translation extends to environments in the obvious manner. Note how the shape of σ^* is the same

Table 1. *Rank 2 properties*

- $\mathsf{t}^{\mathsf{int}} \in \mathit{Prop}_0(\mathsf{int})$
- $\dfrac{\varphi_1 \in \mathit{Prop}_0(\sigma) \quad \varphi_2 \in \mathit{Prop}_0(\tau) \quad \kappa \in \mathit{Labels}(\sigma \to \tau)}{(\varphi_1 \to \varphi_2, \kappa) \in \mathit{Prop}_0(\sigma \to \tau)}$
- $\dfrac{\varphi \in \mathit{Prop}_0(\sigma)}{\varphi \in \mathit{Prop}_1(\sigma)}$
- $\dfrac{\varphi_i \in \mathit{Prop}_0(\sigma), i \in I}{\bigwedge_{i \in I} \varphi_i \in \mathit{Prop}_1(\sigma)}$
- $\dfrac{\varphi \in \mathit{Prop}_0(\sigma)}{\varphi \in \mathit{Prop}_2(\sigma)}$
- $\dfrac{\varphi_1 \in \mathit{Prop}_1(\sigma) \quad \varphi_2 \in \mathit{Prop}_2(\tau), \quad \kappa \in \mathit{Labels}(\sigma \to \tau)}{(\varphi_1 \to \varphi_2, \kappa) \in \mathit{Prop}_2(\sigma \to \tau)}$

as that of σ: if σ has n arrows, so has σ^*. We will exploit this property in the inference algorithm in Section 6 when generating fresh properties for variables.

Example: The control flow properties of the term $\lambda^3 x^{(\mathsf{int}\to\mathsf{int})}.\ \lambda^4 y^{\mathsf{int}}.\ y$ are described by the property

$$((\mathsf{t}^{\mathsf{int}} \to \mathsf{t}^{\mathsf{int}}, \xi) \to (\mathsf{t}^{\mathsf{int}} \to \mathsf{t}^{\mathsf{int}}, \{4\}), \{3\}).$$

which states that the value of the expression is the lambda abstraction labelled 3 and that no matter what function it receives as argument it will return the lambda expression labelled 4 as result.

4. Inference rules for control flow

4.1. ℓPCF

The language of study, ℓPCF, called "labelled call-by-value PCF", is an instrumented version of vPCF, with lambda abstractions and recursive functions annotated by labels. Such labelled functions are the control-flow properties of interest. Values in ℓPCF are integers and labelled functions.

$$\begin{aligned} v &::= n \mid \lambda^{\ell} x^{\sigma}.\ e \mid \mathsf{fun}^{\ell}\ f^{\sigma\to\tau}\ (x^{\sigma}) = e \\ e &::= v \mid x \mid e_1 e_2 \mid \mathsf{op}\ e \mid \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \end{aligned}$$

We will assume that labels in the expression to be analysed are disjoint from the set of variables. Also we assume the variable convention of (Barendregt 1984) *i.e.*, for any expression, the set of its free variables is disjoint from the set of its bound variables.

Just as an ordinary environment, Γ, in the specification of vPCF maps variables to types, a property environment maps variables to properties. The sum of two property environments, π_1 and π_2, denoted by $\pi_1 + \pi_2$, is defined to be the property environment π, where: $\mathit{Dom}(\pi) = \mathit{Dom}(\pi_1) \cup \mathit{Dom}(\pi_2)$, and:

$$\pi(x) = \begin{cases} \pi_1(x) \wedge \pi_2(x), & \text{if } x \in \mathit{Dom}(\pi_1) \cap \mathit{Dom}(\pi_2) \\ \pi_1(x), & \text{if if } x \in \mathit{Dom}(\pi_1) \text{ and } x \notin \mathit{Dom}(\pi_2) \\ \pi_2(x), & \text{if } x \in \mathit{Dom}(\pi_2) \text{ and } x \notin \mathit{Dom}(\pi_1) \end{cases}$$

The disjoint sum of two property environments, π_1 and π_2, denoted by $\pi_1 \oplus \pi_2$, is defined

provided $Dom(\pi_1)$ and $Dom(\pi_2)$ are disjoint, in which case it is $\pi_1 + \pi_2$ (with the first case absent since $Dom(\pi_1)$ and $Dom(\pi_2)$ are disjoint).

Before giving the inference rules for polyvariant control-flow analysis, we give an axiomatisation of an inclusion relation between properties. Intuitively, the inclusion relation expresses when one control flow property is more approximate than another. The situation arises, *e.g.*, in a conditional expression, where the analysis for each branch of the conditional may yield different sets of functions, but each set must be coerced to a common superset. A similar situation arises in applications.

4.2. *Property orderings*

Formally, we axiomatise propositions of the form, $\varphi_1 \leq \varphi_2$, where φ_1 and φ_2 are control flow properties.

Ordering on properties

- $\varphi \leq \varphi$
- $\dfrac{\varphi_1 \leq \varphi_2 \quad \varphi_2 \leq \varphi_3}{\varphi_1 \leq \varphi_3}$
- $\dfrac{\psi_1 \leq \varphi_1 \quad \varphi_2 \leq \psi_2 \quad \kappa_1 \subseteq \kappa_2}{(\varphi_1 \to \varphi_2, \kappa_1) \leq (\psi_1 \to \psi_2, \kappa_2)}$
- $\dfrac{\forall \varphi \in \{\varphi_j \mid j \in J\}.\ \exists \psi \in \{\psi_i \mid i \in I\}.\ \psi \leq \varphi}{(\bigwedge_{i \in I} \psi_i) \leq (\bigwedge_{j \in J} \varphi_j)} \quad \varphi_j \in Prop_0(\sigma), \quad \psi_i \in Prop_0(\sigma)$

Note that the following can be easily derived from the above ordering on properties:

- $\varphi_1 \wedge \varphi_2 \leq \varphi_1$
- $\varphi_1 \wedge \varphi_2 \leq \varphi_2$
- $\dfrac{\varphi \leq \varphi_i \quad i \in I}{\varphi \leq \bigwedge_{i \in I} \varphi_i}$

We now show several properties of the $\leq$ ordering. First, we define the notion of substitution germane to this paper.

Definition 4.1 A *substitution* is a map from label variables to finite sets of labels.

We write $[\,]$ for the empty substitution. A substitution S is lifted to properties as follows: $S(\mathsf{t}^{\mathsf{int}}) = \mathsf{t}^{\mathsf{int}}$; and, for any property φ in which all occurrences of labels are label variables, $S(\varphi)$ is defined in the obvious manner. We say that property φ is *ground* if it does not contain any occurrences of property variables. A property environment π is ground if $\pi(x)$ is ground for all $x \in Dom(\pi)$. The next three propositions relate ground properties, φ, at rank i for $i = 0, 1, 2$ to their underlying types.

Proposition 4.2 Let $\varphi \in Prop_0(\sigma)$. Then there exists substitution S with $S(\sigma^*) = \varphi$.

Proof. By induction on structure of properties at rank 0.

- Case of $\mathsf{t}^{\mathsf{int}}$: Need substitution S such that $S(\mathsf{int}^*) = \mathsf{t}^{\mathsf{int}}$. Choose S to be $[\,]$.
- Case of $(\varphi_1 \to \varphi_2, \kappa)$: Then, $\varphi_1 \in Prop_0(\sigma)$ and $\varphi_2 \in Prop_0(\tau)$ and $\kappa \in Labels(\sigma \to \tau)$. Therefore, by the induction hypothesis on φ_1 and on φ_2, there exist substitutions, S_1 and S_2, such that $S_1(\sigma^*) = \varphi_1$ and $S_2(\tau^*) = \varphi_2$. Need substitution S such that $S(\sigma \to \tau)^* = (\varphi_1 \to \varphi_2, \kappa)$. That is, $S(\sigma^* \to \tau^*, \xi) = (\varphi_1 \to \varphi_2, \kappa)$, where ξ is the label variable, chosen fresh for $(\sigma \to \tau)^*$. Let S_3 be the substitution, $[\xi \mapsto \kappa]$. Then choose the required substitution S to be $S_3 \circ S_2 \circ S_1$. □

Proposition 4.3 Let $\varphi \in \mathit{Prop}_1(\sigma)$. Then there exists substitution S with $\varphi \leq S(\sigma^*)$.

Proof. There are two cases: If $\varphi \in \mathit{Prop}_0(\sigma)$, then by Proposition 4.2 there exists substitution S such that $S(\sigma^*) = \varphi$, hence $\varphi \leq S(\sigma^*)$. Otherwise, $\varphi = (\bigwedge_{i \in I} \varphi_i)$. Now, for every $i \in I$, $\varphi_i \in \mathit{Prop}_0(\sigma)$. Hence, for any i, by Proposition 4.2 there exists substitution S such that $S(\sigma^*) = \varphi_i$; thus for any i, we have $\varphi_i \leq S(\sigma^*)$, hence by the ordering on properties, $(\bigwedge_{i \in I} \varphi_i) \leq S(\sigma^*)$. □

Proposition 4.4 Let $\varphi \in \mathit{Prop}_2(\sigma)$. Then there exists substitution S with $S(\sigma^*) \leq \varphi$.

Proof. There are two cases: If $\varphi \in \mathit{Prop}_0(\sigma)$, then by Proposition 4.2, there exists substitution S such that $S(\sigma^*) = \varphi$, hence $S(\sigma^*) \leq \varphi$. Otherwise, $\varphi = (\varphi_1 \rightarrow \varphi_2, \kappa)$, where $\varphi_1 \in \mathit{Prop}_1(\sigma)$, $\varphi_2 \in \mathit{Prop}_2(\tau)$ and $\kappa \in \mathit{Labels}(\sigma \rightarrow \tau)$. By Proposition 4.3, there exists substitution S_1 such that $\varphi_1 \leq S_1(\sigma^*)$. By the induction hypothesis on φ_2, there exists substitution S_2 such that $S_2(\tau^*) \leq \varphi_2$. Let S_3 be the substitution $[\xi \mapsto \kappa]$. Choose the required substitution $S = S_3 \circ S_2 \circ S_1$. Then by ordering on properties, $S(\sigma^* \rightarrow \tau^*, \xi) \leq (\varphi_1 \rightarrow \varphi_2, \kappa)$. □

Ordering on environments

The ordering on properties is extended to environments in a pointwise manner.

Definition 4.5

Let π, η be rank 1 property environments. Then, say that $\eta \leq \pi$, provided for all $x \in \mathit{Dom}(\pi)$, it is the case that $x \in \mathit{Dom}(\eta)$ and $\eta(x) \leq \pi(x)$.

Definition 4.6

Let π, η be rank 1 property environments and φ, ψ be rank 2 properties. Then, say that $\langle \pi,\ \varphi \rangle \leq \langle \eta,\ \psi \rangle$ provided $\eta \leq \pi$, and $\varphi \leq \psi$.

Proposition 4.7

Let η, π_1, π_2 be rank 1 property environments. Suppose $\eta \leq \pi_i$, for $i = 1, 2$. Then, $\eta \leq \pi_1 + \pi_2$.

Proof. Suppose $x \in \mathit{Dom}(\pi_1 + \pi_2)$. Then clearly $x \in \mathit{Dom}(\eta)$. We need to show $\eta(x) \leq (\pi_1 + \pi_2)(x)$, given $\eta(x) \leq \pi_1(x)$ and $\eta(x) \leq \pi_2(x)$. Suppose $\pi_1(x) = \varphi_1$ and $\pi_2(x) = \varphi_2$. Then $\eta(x) \leq \varphi_1 \wedge \varphi_2$ follows by ordering on properties. □

4.3. *A property system for polyvariant closure analysis*

Now we are in a position to explain the property system for polyvariant control-flow analysis, given in Table 2. Judgements in the system have the form, $\pi \vdash e^{\sigma} : \varphi$, *i.e.*, in rank 1 property environment π, expression e of type σ has rank 2 property φ. An invariant that the property system must satisfy is that the erasure of the property annotations from π and φ must yield a judgement typable in the underlying type system; in particular, the erasure of φ must yield σ. This can be easily formalised, but we choose not to do so in this paper. Rule **Var** is applied at the leaves of the derivation tree. Since property environments are rank 1, *i.e.*, the property associated with each variable in the

Table 2. *Property System for Polyvariant Control-Flow Analysis.*

$$\textbf{Var}\ \ \pi \oplus \{x : \bigwedge_{i\in I} \varphi_i\} \vdash x^\sigma : \varphi_j \quad j \in I$$

$$\textbf{Int}\ \ \pi \vdash n^{\mathsf{int}} : \mathtt{t}^{\mathsf{int}}$$

$$\textbf{Lam}\ \ \frac{\pi \oplus \{x : \bigwedge_{i\in I} \varphi_i\} \vdash e^\tau : \varphi \quad \ell \in L}{\pi \vdash \lambda^\ell x^\sigma.\, e^\tau : ((\bigwedge_{i\in I} \varphi_i) \to \varphi, L)}$$

$$\textbf{App}\ \ \frac{\pi \vdash e_1^{\sigma\to\tau} : ((\bigwedge_{i\in I} \varphi_i) \to \varphi, \kappa) \quad \forall i \in I :\ \pi \vdash e_2^\sigma : \varphi_i' \quad \forall i \in I :\ \varphi_i' \le \varphi_i}{\pi \vdash e_1^{\sigma\to\tau} e_2^\sigma : \varphi}$$

$$\textbf{If}\ \ \frac{\pi \vdash e_1^{\mathsf{int}} : \mathtt{t}^{\mathsf{int}} \quad \pi \vdash e_2^\sigma : \varphi_2 \quad \pi \vdash e_3^\sigma : \varphi_3 \quad \varphi_2 \le \varphi \quad \varphi_3 \le \varphi}{\pi \vdash \mathsf{if}\ e_1^{\mathsf{int}}\ \mathsf{then}\ e_2^\sigma\ \mathsf{else}\ e_3^\sigma : \varphi}$$

$$\textbf{Fun}\ \ \frac{\pi \oplus \{f : \bigwedge_{i\in I}(\varphi_i \to \varphi_i', L_i), x : \bigwedge_{j\in J} \theta_j\} \vdash e^\tau : \theta \quad \forall i \in I : ((\bigwedge_{j\in J} \theta_j) \to \theta, L) \le (\varphi_i \to \varphi_i', L_i) \quad \ell \in L}{\pi \vdash \mathsf{fun}^\ell\, f^{\sigma\to\tau}\, (x^\sigma) = e^\tau : ((\bigwedge_{j\in J} \theta_j) \to \theta, L)}$$

$$\textbf{Op}\ \ \frac{\pi \vdash e^{\mathsf{int}} : \mathtt{t}^{\mathsf{int}}}{\pi \vdash \mathsf{op}\ e^{\mathsf{int}} : \mathtt{t}^{\mathsf{int}}}$$

environment is a conjunction of rank 0 properties, we can (non-deterministically) choose an appropriate conjunct as property of the variable. In rule **Lam**, the bound variable, x, may have multiple occurrences in the body, e, of the lambda abstraction labelled ℓ. Since we are interested in polyvariance, therefore, to represent all the hypotheses made on x, we choose to give it an intersection property. In particular, if there is no occurrence of x in the body, then x can be given any property as long as the erasure of x's property annotation yields its underlying type, σ. Note that ℓ must be contained in the set of possible functions that the lambda abstraction can evaluate to. In rule **App**, if the function part returns a rank 2 type of the form $((\bigwedge_{i\in I} \varphi_i) \to \varphi, \kappa)$, then the argument part must have properties φ_i' for each $i \in I$, such that φ_i' is a "sub-property" of φ_i, according to the ordering on properties in Section 4. In rule **If**, the branches of the conditional may yield different properties, φ_2, φ_3. But since we are interested in specifying a *static* analysis, both φ_2 and φ_3 must be a sub-property of a common property φ according to the ordering on properties. In Rule **Fun**, each recursive function invocation can assign different properties to a recursive function f with label ℓ. Hence f is given the intersection type $\bigwedge_{i\in I}(\varphi_i \to \varphi_i', L_i)$ in the antecedent. As in rule **Lam**,

we require that ℓ be contained in the set of possible functions the recursive function can evaluate to (and also in each L_i).

Note how the property system is syntax-directed; there is no explicit rule for subsumption – instead subsumption is inlined in the rules it can occur, namely, rules **App**, **If** and **Fun**. This is done for technical convenience, namely, to facilitate proofs by structural induction. One can show the following propositions for the property system.

Proposition 4.8 (Weakening)
Let $\pi \vdash e : \varphi$ be derivable. Then for any property environment π', it is the case that $\pi + \pi' \vdash e : \varphi$.

Proposition 4.9 (Subproperty)
Let $\pi \vdash e : \varphi$ be derivable. Then $\pi' \vdash e : \varphi'$ is derivable provided $\pi' \leq \pi$ and $\varphi \leq \varphi'$.

The proofs for both propositions are easy inductions on the height of the derivation tree of e.

Example: Consider our running example from Section 2:

$$T = (\lambda^1 g^{(\mathsf{int}\to\mathsf{int})\to(\mathsf{int}\to\mathsf{int})}.\ g\ (g\ (\lambda^2 v^{\mathsf{int}}.\ v)))\ \ (\lambda^3 x^{(\mathsf{int}\to\mathsf{int})}.\ \lambda^4 y^{\mathsf{int}}.\ y)$$

Let

$$\varphi = ((\mathtt{t}^{\mathsf{int}} \to \mathtt{t}^{\mathsf{int}}, \{4\}) \to (\mathtt{t}^{\mathsf{int}} \to \mathtt{t}^{\mathsf{int}}, \{4\}), \{3\})$$

and

$$\varphi' = ((\mathtt{t}^{\mathsf{int}} \to \mathtt{t}^{\mathsf{int}}, \{2\}) \to (\mathtt{t}^{\mathsf{int}} \to \mathtt{t}^{\mathsf{int}}, \{4\}), \{3\})$$

We first show a typing derivation of the function part, $\lambda^1 g^{(\mathsf{int}\to\mathsf{int})\to(\mathsf{int}\to\mathsf{int})}.\ g\ (g\ (\lambda^2 v^{\mathsf{int}}.\ v))$, in stages.

$$\textbf{(A)}\ \frac{\{g : \varphi \wedge \varphi'\} \vdash g : \varphi' \qquad \frac{\{g:\varphi\wedge\varphi', v:\mathtt{t}^{\mathsf{int}}\} \vdash v:\mathtt{t}^{\mathsf{int}}}{\{g:\varphi\wedge\varphi'\} \vdash \lambda^2 v.\ v:(\mathtt{t}^{\mathsf{int}}\to\mathtt{t}^{\mathsf{int}},\{2\})}}{\{g : \varphi \wedge \varphi'\} \vdash g(\lambda^2 v.\ v) : (\mathtt{t}^{\mathsf{int}} \to \mathtt{t}^{\mathsf{int}}, \{4\})}$$

From **(A)** using rule **App**,

$$\textbf{(B)}\ \frac{\{g : \varphi \wedge \varphi'\} \vdash g : \varphi \qquad \{g : \varphi \wedge \varphi'\} \vdash g(\lambda^2 v.\ v) : (\mathtt{t}^{\mathsf{int}} \to \mathtt{t}^{\mathsf{int}}, \{4\})}{\{g : \varphi \wedge \varphi'\} \vdash g(g(\lambda^2 v.\ v)) : (\mathtt{t}^{\mathsf{int}} \to \mathtt{t}^{\mathsf{int}}, \{4\})}$$

From **(B)** using rule **Lam**,

$$\textbf{(C)}\ \frac{\{g : \varphi \wedge \varphi'\} \vdash g(g(\lambda^2 v.\ v)) : (\mathtt{t}^{\mathsf{int}} \to \mathtt{t}^{\mathsf{int}}, \{4\})}{\varnothing \vdash \lambda^1 g(g(\lambda^2 v.\ v)).\ : (\varphi \wedge \varphi' \to (\mathtt{t}^{\mathsf{int}} \to \mathtt{t}^{\mathsf{int}}, \{4\}), \{1\})}$$

For the argument part, $\lambda^3 x.\ \lambda^4 y.\ y$, note that using rule **Lam**,

$$\textbf{(D)}\ \frac{\{x : (\mathtt{t}^{\mathsf{int}} \to \mathtt{t}^{\mathsf{int}}, \{2\})\} \vdash \lambda^4 y.\ y : (\mathtt{t}^{\mathsf{int}} \to \mathtt{t}^{\mathsf{int}}, \{4\})}{\varnothing \vdash \lambda^3 x.\ \lambda^4 y.\ y : \varphi'}$$

and

$$\textbf{(E)}\ \frac{\{x : (\mathtt{t}^{\mathsf{int}} \to \mathtt{t}^{\mathsf{int}}, \{4\})\} \vdash \lambda^4 y.\ y : (\mathtt{t}^{\mathsf{int}} \to \mathtt{t}^{\mathsf{int}}, \{4\})}{\varnothing \vdash \lambda^3 x.\ \lambda^4 y.\ y : \varphi}$$

Hence from **(D)**, **(E)** and **(C)**, using rule **App**,

$$\varnothing \vdash T : (\mathtt{t}^{\mathsf{int}} \to \mathtt{t}^{\mathsf{int}}, \{4\})$$

5. Operational semantics and correctness

Table 3 gives the small-step operational semantics of our language. In the operational semantics, the notation $e[x \mapsto v]$ denotes the capture-avoiding substitution of all free occurrences of the variable x by the value v in expression e. The operational semantics show reductions of simple redexes. They lift to arbitrary terms via

$$\frac{e \to e'}{E[e] \to E[e']}$$

where E denotes evaluation contexts (Wright and Felleisen 1991) specified by the BNF:

$$E ::= [\,] \mid (E\ e) \mid (v\ E) \mid \mathsf{if}\ E\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mid \mathsf{succ}\ E \mid \mathsf{pred}\ E$$

Using the operational semantics, we can show type soundness. This requires proving a

Table 3. *Small-step operational semantics*

$(\lambda^{\ell} x.\ e)\ v$	$\to$	$e[x \mapsto v]$
$(\mathsf{fun}^{\ell}\ f\ (x) = e)\ v$	$\to$	$e[f \mapsto (\mathsf{fun}^{\ell}\ f\ (x) = e)][x \mapsto v]$
$(\mathsf{succ}\ n)$	$\to$	$(n+1)$
$(\mathsf{pred}\ 0)$	$\to$	0
$(\mathsf{pred}\ (n+1))$	$\to$	n
$(\mathsf{if}\ 0\ \mathsf{then}\ e\ \mathsf{else}\ e')$	$\to$	e
$(\mathsf{if}\ (n+1)\ \mathsf{then}\ e\ \mathsf{else}\ e')$	$\to$	e'

substitution lemma.

5.1. *Substitution lemma*

Lemma 5.1
Suppose that $\pi \oplus \{x : \bigwedge_{i \in I} \varphi_i\} \vdash e : \varphi$ and for all $i \in I$, $\pi \vdash v : \varphi'_i$ where $\varphi'_i \leq \varphi_i$. Then, there exists φ' such that $\pi \vdash e[x \mapsto v] : \varphi'$ and $\varphi' \leq \varphi$.

Proof. We go by induction on the height of the derivation tree of e and by case analysis on the final step. The following are the interesting cases, the rest being routine. In fact, the case for recursive functions mirrors the case for lambda abstraction.

- Case of $\pi \oplus \{x : \bigwedge_{i \in I} \varphi_i\} \vdash x' : \varphi$: We have two cases: **Case(a):** $x' = x$. Then $\pi \oplus \{x : \bigwedge_{i \in I} \varphi_i\} \vdash x : \varphi_j$, where $j \in I$. Assume that for all $i \in I$, $\pi \vdash v : \varphi'_i$, where $\varphi'_i \leq \varphi_i$. In particular, since $j \in I$, we obtain $\pi \vdash v : \varphi'_j$, where $\varphi'_j \leq \varphi_j$. That is, $\pi \vdash x[x \mapsto v] : \varphi'_j$ and $\varphi'_j \leq \varphi_j$. **Case(b):** $x' \neq x$. Then $x' \in Dom(\pi)$ and $\pi \vdash x' : \varphi$. Thus, $\pi \vdash x'[x \mapsto v] : \varphi$ and $\varphi \leq \varphi$.
- Case of $\pi \oplus \{x : \bigwedge_{i \in I} \varphi_i\} \vdash \lambda^{\ell} x'.\ e : ((\bigwedge_{j \in J} \varphi'_j) \to \varphi', L)$: Assume that $x' \neq x$. (If

$x' = x$, rename the bound variable x' to x''). For all $i \in I$, let $\pi \vdash v : \varphi'_i$ such that $\varphi'_i \leq \varphi_i$. We must show that there exists θ such that $\pi \vdash (\lambda^\ell x'.\ e)[x \mapsto v] : \theta$ and $\theta \leq ((\bigwedge_{j\in J} \varphi'_j) \to \varphi', L)$. From the derivation tree of $\lambda^\ell x'.\ e$, it must be the case that:

$$\textbf{(i)}\ \pi \oplus \{x : \bigwedge_{i\in I} \varphi_i\} \oplus \{x' : \bigwedge_{j\in J} \varphi'_j\} \vdash e : \varphi' \text{ and } \ell \in L.$$

By the induction hypothesis on the derivation tree for e, there exists θ' such that:

$$\textbf{(ii)}\ \pi \oplus \{x' : \bigwedge_{j\in J} \varphi'_j\} \vdash e[x \mapsto v] : \theta' \ \text{ and } \ \textbf{(iii)}\ \theta' \leq \varphi'.$$

Hence, from **(ii)**, using the fact that $\ell \in L$, $\pi \vdash \lambda^\ell x'.\ e[x \mapsto v] : ((\bigwedge_j \varphi'_j) \to \theta', L)$, *i.e.*, $\pi \vdash (\lambda^\ell x'.\ e)[x \mapsto v] : ((\bigwedge_j \varphi'_j) \to \theta', L)$. Now choose $\theta = ((\bigwedge_j \varphi'_j) \to \theta', L)$. Clearly, using **(iii)**, $\theta \leq ((\bigwedge_j \varphi'_j) \to \varphi', L)$.

- Case of $\pi \oplus \{x : \bigwedge_{i\in I} \varphi_i\} \vdash e_1 e_2 : \varphi'$: Assume for all $i \in I$, $\pi \vdash v : \varphi'_i$ where $\varphi'_i \leq \varphi_i$. We must show there exists θ' such that $\pi \vdash (e_1 e_2)[x \mapsto v] : \theta'$ and $\theta' \leq \varphi'$. From the derivation tree of $e_1 e_2$, it must be the case that: **(i)** $\pi \oplus \{x : \bigwedge_{i\in I} \varphi_i\} \vdash e_1 : ((\bigwedge_{j\in J} \varphi'_j) \to \varphi', \kappa)$

$$\textbf{(ii)}\ \ \pi \oplus \{x : \bigwedge_{i\in I} \varphi_i\} \vdash e_2 : \varphi''_j, \text{ for each } j \in J \quad \textbf{(iii)}\ \ \varphi''_j \leq \varphi'_j, \text{ for each } j \in J$$

By the induction hypothesis on derivation tree of e_1, there exists θ with shape $((\bigwedge_{k\in K} \theta'_k) \to \theta', \mu)$, such that:

$$\textbf{(iv)}\ \ \pi \vdash e_1[x \mapsto v] : ((\bigwedge_{k\in K} \theta'_k) \to \theta', \mu) \ \text{ with}$$

$$\textbf{(v)}\ ((\bigwedge_{k\in K} \theta'_k) \to \theta', \mu) \leq ((\bigwedge_{j\in J} \varphi'_j) \to \varphi', \kappa).$$

From **(v)**, by contravariance, $(\bigwedge_{j\in J} \varphi'_j) \leq (\bigwedge_{k\in K} \theta'_k)$. Hence, by the ordering on properties, for all $k \in K$, there exists $j_k \in J$, such that **(vi)** $\varphi'_{j_k} \leq \theta'_k$. From **(v)**, by covariance, **(vii)** $\theta' \leq \varphi'$. By the induction hypothesis on derivation tree of e_2, for all $j \in J$,

$$\textbf{(viii)}\ \ \pi \vdash e_2[x \mapsto v] : \theta''_j, \ \text{ such that } \ \textbf{(ix)}\ \theta''_j \leq \varphi''_j.$$

To show the result, we need to demonstrate using **(iv)** and **(viii)** that for all $k \in K$, $\theta''_{j_k} \leq \theta'_k$. Accordingly, consider any pair (k, j_k) as in **(vi)** where $k \in K$. Then using **(ix)**, **(iii)** and **(vi)** we have, $\theta''_{j_k} \leq \varphi''_{j_k} \leq \varphi'_{j_k} \leq \theta'_k$. Hence by the ordering on properties, **(x)** $\theta''_{j_k} \leq \theta'_k$ for all $k \in K$. Now using **(iv)**, **(viii)** and **(x)** and using the idempotence of $\wedge$ we have, $\pi \vdash e_1[x \mapsto v] e_2[x \mapsto v] : \theta'$, *i.e.*, $\pi \vdash (e_1 e_2)[x \mapsto v] : \theta'$. Now the lemma holds by appealing to **(vii)**. □

5.2. *Type soundness*

To prove type soundness, we proceed in two steps: first we prove a type soundness result for simple redexes defined by the operational semantics in Table 3. Next, we lift this result to arbitrary terms.

Lemma 5.2 Let $\pi \vdash e_1 : \varphi$ and $e_1 \to e_2$, where e_1 and e_2 are the left-hand sides and right-hand sides respectively of the operational semantics in Table 3. Then there exists φ' such that $\pi \vdash e_2 : \varphi'$ and $\varphi' \leq \varphi$.

Proof. We go by cases on the reduction $e_1 \to e_2$ in Table 3. The interesting cases are the ones for application – the other cases being routine.

- Case of $(\lambda^\ell x.\ e)v \to e[x \mapsto v]$: Then the derivation is $\pi \vdash (\lambda^\ell x.\ e)v : \varphi$. Hence:

$$\textbf{(i)}\ \pi \vdash \lambda^\ell x.\ e : ((\bigwedge_{i\in I} \varphi_i) \to \varphi, L) \quad \textbf{(ii)}\ \pi \vdash v : \varphi'_i,\ \text{for each } i \in I$$

$$\textbf{(iii)}\ \varphi'_i \leq \varphi_i,\ \text{for each } i \in I$$

From **(i)**, it must be the case that **(iv)** $\pi \oplus \{x : \bigwedge_{i\in I} \varphi_i\} \vdash e : \varphi$. Hence from **(iv)**, **(ii)** and **(iii)**, using Lemma 5.1, there exists φ' such that $\pi \vdash e[x \mapsto v] : \varphi'$ and $\varphi' \leq \varphi$.

- Case of $(\mathsf{fun}^\ell\, f\,(x) = e)v \to e[f \mapsto (\mathsf{fun}^\ell\, f\,(x) = e)][x \mapsto v]$: Then the derivation is $\pi \vdash (\mathsf{fun}^\ell\, f\,(x) = e)v : \varphi$. Hence

$$\textbf{(i)}\ \pi \vdash \mathsf{fun}^\ell\, f\,(x) = e : ((\bigwedge_{j\in J} \theta_j) \to \varphi, L) \quad \textbf{(ii)}\ \pi \vdash v : \theta'_j,\ \text{for each } j \in J$$

$$\textbf{(iii)}\ \theta'_j \leq \theta_j,\ \text{for each } j \in J$$

From **(i)**, it must be the case that

$$\textbf{(iv)}\ \pi \oplus \{f : \bigwedge_{i\in I} (\varphi_i \to \varphi'_i, L_i), x : (\bigwedge_{j\in J} \theta_j)\} \vdash e : \varphi \quad \text{and}$$

$$\textbf{(v)}\ ((\bigwedge_{j\in J} \theta_j) \to \varphi, L) \leq (\varphi_i \to \varphi'_i, L_i),\ \text{for each } i \in I$$

Hence from **(iv)**, **(i)** and **(v)**, applying Lemma 5.1, there exists φ' such that

$$\textbf{(vi)}\ \pi \oplus \{x : (\bigwedge_{j\in J} \theta_j)\} \vdash e[f \mapsto \mathsf{fun}^\ell\, f\,(x) = e] : \varphi' \quad \text{and } \textbf{(vii)}\ \varphi' \leq \varphi.$$

Now from **(vi)**, **(ii)** and **(iii)**, applying Lemma 5.1 again, there exists φ'' such that **(viii)** $\pi \vdash e[f \mapsto \mathsf{fun}^\ell\, f\,(x) = e][x \mapsto v] : \varphi''$, and **(ix)** $\varphi'' \leq \varphi'$. Hence from **(ix)** and **(vii)**, $\varphi'' \leq \varphi$. □

Theorem 5.3 (Type Soundness Theorem)
Let $\pi \vdash e : \varphi$ and $e \to e'$. Then there exists φ' such that $\pi \vdash e' : \varphi'$ and $\varphi' \leq \varphi$.

Proof. By a simple induction on the structure of evaluation contexts, $e = E[e_1]$, where $e_1 \to e_2$ via one of the rules in Table 3, and $e' = E[e_2]$. Now an induction on the structure of evaluation contexts using Lemma 5.2 completes the proof. □

6. An inference algorithm based on ranked properties

In this section we present an inference algorithm for the property system in Table 2. For an expression e^σ, algorithm $\mathcal{I}$ computes a triple $\langle \pi,\ \varphi,\ C \rangle$, where π is the inferred rank

1 property environment for the free variables of e^σ, φ is the inferred rank 2 property, and C is a set of constraints where each constraint in C has the form, $\xi \subseteq \xi'$ or $L \subseteq \xi$ for some label variables ξ, ξ' and label set L. The algorithm $\mathcal{I}$ is specified in Table 4 in a bottom-up manner, reminiscent of Damas's Algorithm T (Damas 1985). We explain the cases for variables, lambda abstractions and applications below.

Table 4. *The inference algorithm, $\mathcal{I}$*

Variables ξ, ξ_i used in the inference algorithm are required to be fresh.

$$
\begin{array}{lcl}
\mathcal{I}(x^\sigma) & = & \textbf{let } \varphi = \sigma^* \textbf{ in } \langle \{x^\sigma : \varphi\},\ \varphi,\ \varnothing \rangle \\[2ex]
\mathcal{I}(n^{\mathsf{int}}) & = & \langle \varnothing,\ \mathtt{t}^{\mathsf{int}},\ \varnothing \rangle \\[2ex]
\mathcal{I}(\lambda^\ell x^\sigma.\, e^\tau) & = & \textbf{let } \langle \pi,\ \varphi,\ C \rangle = \mathcal{I}(e^\tau) \textbf{ in} \\
 & & \quad \langle \pi',\ ((\bigwedge_{i\in I} \varphi_i) \to \varphi, \xi),\ C \cup \{\{\ell\} \subseteq \xi\} \rangle \quad \textbf{if } \pi = \pi' \oplus \{x^\sigma : \textstyle\bigwedge_{i\in I} \varphi_i\} \\
 & & \quad \langle \pi,\ (\psi \to \varphi, \xi),\ C \cup \{\{\ell\} \subseteq \xi\} \rangle \quad \textbf{if } \psi = \sigma^* \textbf{ and } x^\sigma \notin \mathit{Dom}(\pi) \\[2ex]
\mathcal{I}(e_1^{\sigma\to\tau} e_2^\sigma) & = & \textbf{let } \langle \pi_1,\ \varphi_1,\ C_1 \rangle = \mathcal{I}(e_1^{\sigma\to\tau}) \\
 & & \quad \langle \pi_2,\ \varphi_2,\ C_2 \rangle = \mathcal{I}(e_2^\sigma) \\
 & & \textbf{in case } \varphi_1 \textbf{ of} \\
 & & \quad ((\bigwedge_{i\in I} \varphi_{1i}) \to \varphi_{12}, \xi) : \\
 & & \qquad \textbf{let } \left\langle \widetilde{\pi}_i,\ \widetilde{\varphi}_i,\ \widetilde{C}_i \right\rangle = \mathit{Rename}(\pi_2, \varphi_2, C_2) \\
 & & \qquad\quad \{\widetilde{\varphi}_i \le \varphi_{1i} \mid i \in I\} \overset{+}{\rightsquigarrow} C' \\
 & & \qquad\quad C = C_1 \cup (\textstyle\bigcup_{i\in I} \widetilde{C}_i) \cup C' \\
 & & \qquad \textbf{in } \langle \pi_1 + \textstyle\sum_{i\in I} \widetilde{\pi}_i,\ \varphi_{12},\ C \rangle \\[2ex]
\mathcal{I}(\mathsf{if}\ e_1^{\mathsf{int}}\ \mathsf{then}\ e_2^\sigma\ \mathsf{else}\ e_3^\sigma) = & & \textbf{let} \\
 & & \quad \langle \pi_1,\ \mathtt{t}^{\mathsf{int}},\ C_1 \rangle = \mathcal{I}(e_1^{\mathsf{int}}) \\
 & & \quad \langle \pi_2,\ \varphi_2,\ C_2 \rangle = \mathcal{I}(e_2^\sigma) \\
 & & \quad \langle \pi_3,\ \varphi_3,\ C_3 \rangle = \mathcal{I}(e_3^\sigma) \\
 & & \quad \varphi = \sigma^* \\
 & & \quad \{\varphi_2 \le \varphi, \varphi_3 \le \varphi\} \overset{+}{\rightsquigarrow} C_4 \\
 & & \quad C = C_1 \cup C_2 \cup C_3 \cup C_4 \\
 & & \textbf{in } \langle \pi_1 + \pi_2 + \pi_3,\ \varphi,\ C \rangle \\[2ex]
\mathcal{I}(\mathsf{fun}^\ell\, f^{\sigma\to\tau}\, (x^\sigma) = e^\tau) & = & \left\{ \begin{array}{l}
\textbf{if } x \notin \mathit{FV}(e) \textbf{ then} \\
\textbf{let} \\
\langle \pi \oplus \{f : \bigwedge_{i\in I} (\varphi_i \to \varphi_i', \xi_i)\}, \delta, C \rangle = \mathcal{I}(e^\tau) \\
\{(\sigma^* \to \delta, \xi) \le \bigwedge_{i\in I} (\varphi_i \to \varphi_i', \xi_i)\} \overset{+}{\rightsquigarrow} C_1 \\
\textbf{in } \langle \pi,\ (\sigma^* \to \delta, \xi),\ C \cup C_1 \cup \{\{\ell\} \subseteq \xi\} \rangle \\
\textbf{else let} \\
\left\langle \pi \oplus \{f : \bigwedge_{i\in I} (\varphi_i \to \varphi_i', \xi_i), x : (\bigwedge_{j\in J} \delta_j)\}, \delta, C \right\rangle = \mathcal{I}(e^\tau) \\
\{((\bigwedge_{j\in J} \delta_j) \to \delta, \xi) \le \bigwedge_{i\in I} (\varphi_i \to \varphi_i', \xi_i)\} \overset{+}{\rightsquigarrow} C_1 \\
\textbf{in } \left\langle \pi,\ ((\bigwedge_{j\in J} \delta_j) \to \delta, \xi),\ C \cup C_1 \cup \{\{\ell\} \subseteq \xi\} \right\rangle
\end{array} \right. \\[2ex]
\mathcal{I}(\mathsf{op}\ e^{\mathsf{int}}) & = & \mathcal{I}(e^{\mathsf{int}})
\end{array}
$$

For a variable, x^σ, the algorithm returns a property $\varphi = \sigma^*$, since the flow must have the same shape as the type σ. For example, if σ is the type $\mathsf{int} \to \mathsf{int}$, then we know that

x can only be bound to functions of type $\mathsf{int} \to \mathsf{int}$. Therefore, x's property must have the shape $(\mathsf{t}^{\mathsf{int}} \to \mathsf{t}^{\mathsf{int}}, \xi)$ where fresh ξ is the placeholder for the set of possible functions that can be bound to x.

For a lambda abstraction, $\lambda^{\ell} x^{\sigma}.\ e^{\tau}$, the algorithm first analyses the body e. Suppose the inferred property for e is φ. Let $FV(e)$ denote the set of free variables in e. If $x \in FV(e)$, then the property φ_i of each occurrence is collected together in the inferred property environment π, so that $\pi(x) = \bigwedge_{i \in I} \varphi_i$. Moreover, each $\varphi_i = \sigma^*$. The property inferred for the entire lambda abstraction is $((\bigwedge_{i \in I} \varphi_i) \to \varphi, \xi)$ where ξ is fresh and $\{\ell\} \subseteq \xi$. If $x \notin FV(e)$, then $x \notin \mathit{Dom}(\pi)$. Then the inferred property for the entire lambda abstraction is $(\psi \to \varphi, \xi)$, where ξ is fresh and $\psi = \sigma^*$ and $\{\ell\} \subseteq \xi$.

For the application $e_1^{\sigma \to \tau} e_2^{\sigma}$, the algorithm first analyses e_1. Suppose the inferred environment is π_1. The inferred property for e_1 must have the shape $((\bigwedge_{i \in I} \varphi_{1i}) \to \varphi_{12}, \xi)$. Next e_2 is analysed. Suppose the inferred environment is π_2 and the inferred property is φ_2. From the inference rule for application in Table 2, we need to make i copies of φ_2 and, for every $i \in I$, "satisfy the ordering" $\widetilde{\varphi}_i \leq \varphi_{1i}$, where $\widetilde{\varphi}_i$ is the i-th copy of φ_2. Once the ordering is satisfied, the algorithm returns the environment $\pi_1 + \Sigma_{i \in I} \pi_i$ and the property φ_{12} for the entire application.

Consider properties φ, φ' in which all labels (if any) are label variables; then we say that substitution S *satisfies* $\varphi \leq \varphi'$ provided applying the substitution to φ, φ' maintains the ordering; *i.e.*, $S(\varphi) \leq S(\varphi')$. (This is lifted to a set of orderings in the obvious manner). How can we satisfy $\widetilde{\varphi}_i \leq \varphi_{1i}$, for every $i \in I$? By inspection of the inference algorithm, first note that if $\mathcal{I}(e^{\sigma \to \tau}) = \langle \pi,\ \varphi,\ C \rangle$, then $\varphi \in \mathit{Prop}_2(\sigma \to \tau)$ and all labels in φ are label variables. Note too the shapes of orderings on properties: these can only be

- $\mathsf{t}^{\mathsf{int}} \leq \mathsf{t}^{\mathsf{int}}$
- $\varphi \leq \bigwedge_{i \in I} \varphi_i$
- $(\varphi_1 \to \varphi_2, \xi_1) \leq (\varphi_3 \to \varphi_4, \xi_2)$.

Now the specification of $\leq$ in Section 4 gives an algorithm for satisfying the set $\{\widetilde{\varphi}_i \leq \varphi_{1i} \mid i \in I\}$: we just "run the rules backwards". We formalise this using the $\rightsquigarrow$ relation below; the main idea is to decompose the set, K, of ordering on properties, into a set, C, of constraints between label variables, *i.e.*, $\langle K, C \rangle \overset{+}{\rightsquigarrow} \langle K', C' \rangle$.

$$
\begin{array}{lcl}
\langle \{\mathsf{t}^{\mathsf{int}} \leq \mathsf{t}^{\mathsf{int}}\} \cup K, C \rangle & \rightsquigarrow & \langle K, C \rangle \\
\langle \{\varphi \leq (\bigwedge_{i \in I} \varphi_i)\} \cup K, C \rangle & \rightsquigarrow & \langle \{\varphi \leq \varphi_i \mid i \in I\} \cup K, C \rangle \\
\langle \{(\varphi_1 \to \varphi_2, \xi_1) \leq (\varphi_3 \to \varphi_4, \xi_2)\} \cup K, C \rangle & \rightsquigarrow & \langle \{\varphi_3 \leq \varphi_1, \varphi_2 \leq \varphi_4\} \cup K, C \cup \{\xi_1 \subseteq \xi_2\} \rangle
\end{array}
$$

Note that the $\rightsquigarrow$ relation as defined above is indeed an *algorithm*, since at every rewrite step either the number of arrows, or the number of intersections, or the size of the constraint set on the left hand side decreases. More formally, we can prove the termination of rewriting by a lexicographic induction on (a, i, s), where a is the number of arrows in K, i is the number of intersections in K, and s is the size of K. Upon termination, we obtain a set of constraints, where each constraint has form $\xi \subseteq \xi'$.

We say that a substitution S is a *solution* of a constraint $\kappa \subseteq \kappa'$, if $S(\kappa) \subseteq S(\kappa')$ holds. (This is lifted to a set of constraints in the obvious manner).

Lemma 6.1 Let $\langle K, C \rangle \rightsquigarrow \langle K', C' \rangle$. For any substitution S, S satisfies K and S is a solution to C iff S satisfies K' and S is a solution to C'.

Proof. By cases on the reduction $\langle K, C\rangle \rightsquigarrow \langle K', C'\rangle$.

— $\langle \{\mathsf{t}^{\mathsf{int}} \leq \mathsf{t}^{\mathsf{int}}\} \cup K, C\rangle \rightsquigarrow \langle K, C\rangle$: Let S satisfy $\{\mathsf{t}^{\mathsf{int}} \leq \mathsf{t}^{\mathsf{int}}\} \cup K$ and let S be a solution to C. Then S satisfies K.
Conversely, let S satisfy K and let S be a solution to C. Then S satisfies $\{\mathsf{t}^{\mathsf{int}} \leq \mathsf{t}^{\mathsf{int}}\} \cup K$.

— $\langle \{\varphi \leq (\bigwedge_{i\in I} \varphi_i)\} \cup K, C\rangle \rightsquigarrow \langle \{\varphi \leq \varphi_i \mid i \in I\} \cup K, C\rangle$: Let S satisfy $\{\varphi \leq (\bigwedge_{i\in I} \varphi_i)\} \cup K$ and let S be a solution to C. Then $S(\varphi) \leq S(\bigwedge_{i\in I} \varphi_i)$ holds. Hence, by ordering on properties it must be that $S(\varphi) \leq S(\varphi_i)$, for every $i \in I$. Hence S satisfies $\{\varphi \leq (\bigwedge_{i\in I} \varphi_i)\} \cup K$.
Conversely, let S satisfy $\{\varphi \leq \varphi_i \mid i \in I\} \cup K$ and let S be a solution to C. That is, S satisfies K, and $S(\varphi) \leq S(\varphi_i)$ for every $i \in I$. But then by ordering on properties, $S(\varphi) \leq (\bigwedge_{i\in I} S(\varphi_i))$, *i.e.*, $S(\varphi) \leq S(\bigwedge_{i\in I}(\varphi_i))$. Hence S satisfies $\{\varphi \leq (\bigwedge_{i\in I}(\varphi_i))\} \cup K$.

— $\langle \{(\varphi_1 \to \varphi_2, \xi_1) \leq (\varphi_3 \to \varphi_4, \xi_2)\} \cup K, C\rangle \rightsquigarrow \langle \{\varphi_3 \leq \varphi_1, \varphi_2 \leq \varphi_4\} \cup K, C \cup \{\xi_1 \subseteq \xi_2\}\rangle$: Let S satisfy $\{(\varphi_1 \to \varphi_2, \xi_1) \leq (\varphi_3 \to \varphi_4, \xi_2)\} \cup K$ and let S be a solution to C. Then $S(\varphi_1 \to \varphi_2, \xi_1) \leq S(\varphi_3 \to \varphi_4, \xi_2)$. Hence by ordering on properties it must be that $S(\varphi_3) \leq S(\varphi_1)$, $S(\varphi_2) \leq S(\varphi_4)$ and $S(\xi_1) \subseteq S(\xi_2)$. Hence S satisfies $\{\varphi_3 \leq \varphi_1, \varphi_2 \leq \varphi_4\} \cup K$ and S is a solution to $C \cup \{\xi_1 \subseteq \xi_2\}$.
Conversely, let S satisfy $\{\varphi_3 \leq \varphi_1, \varphi_2 \leq \varphi_4\} \cup K$ and let S be a solution to $C \cup \{\xi_1 \subseteq \xi_2\}$. That is, S satisfies K and $S(\varphi_3) \leq S(\varphi_1)$ and $S(\varphi_2) \leq S(\varphi_4)$. Moreover, $S(\xi_1) \subseteq S(\xi_2)$. Hence by ordering on properties, $S(\varphi_1 \to \varphi_2, \xi_1) \leq S(\varphi_3 \to \varphi_4, \xi_2)$, *i.e.*, S satisfies $\{(\varphi_1 \to \varphi_2, \xi_1) \leq (\varphi_3 \to \varphi_4, \xi_2)\} \cup K$ and S is a solution to C.

□

A consequence of Lemma 6.1 is that it holds when $\rightsquigarrow$ is replaced by $\overset{+}{\rightsquigarrow}$, where $\overset{+}{\rightsquigarrow}$ denotes repeated rewriting using $\rightsquigarrow$. More specifically, an initial set, K, of orderings on properties can be rewritten to a set of constraints, C, on label variables; substitution S satisfies K iff S is a solution to C.

Lemma 6.2 Let $\langle K, \varnothing\rangle \overset{+}{\rightsquigarrow} \langle \varnothing, C\rangle$. Then S satisfies K iff S is a solution to C.

Proof. Directly from Lemma 6.1, noting that any S that satisfies K is a solution to the empty set of constraints and satisfies the empty set of orderings between properties. □

In the sequel we will abbreviate $\langle K, \varnothing\rangle \overset{+}{\rightsquigarrow} \langle \varnothing, C\rangle$ as $K \overset{+}{\rightsquigarrow} C$. Note that by inspection of algorithm $\mathcal{I}$, for any expression e, if $\mathcal{I}(e) = \langle \pi,\ \varphi,\ C\rangle$, then all constraints in C have the form $\kappa \subseteq \xi$. Such a set of constraints always admits a solution computed by the usual transitive closure algorithm.

6.1. *Example*

Consider our example from Section 2:

$$T = (\lambda^1 g^{(\mathsf{int}\to\mathsf{int})\to(\mathsf{int}\to\mathsf{int})}.\ g\ (g\ (\lambda^2 v^{\mathsf{int}}.\ v)))\ \ (\lambda^3 x^{(\mathsf{int}\to\mathsf{int})}.\ \lambda^4 y^{\mathsf{int}}.\ y)$$

We show the result of the inference algorithm of Table 4 applied to T. First, consider the argument part, $\lambda^3 x^{(\mathsf{int}\to\mathsf{int})}.\ \lambda^4 y^{\mathsf{int}}.\ y$. We have:

$$\begin{array}{lcl}
\mathcal{I}(y) & = & \langle \{y : \mathsf{t}^{\mathsf{int}}\},\ \mathsf{t}^{\mathsf{int}},\ \varnothing \rangle \\
\mathcal{I}(\lambda^4 y.\ y) & = & \langle \varnothing,\ (\mathsf{t}^{\mathsf{int}} \to \mathsf{t}^{\mathsf{int}}, \xi_1),\ \{\{4\} \subseteq \xi_1\} \rangle \\
\mathcal{I}(\lambda^3 x.\ \lambda^4 y.\ y) & = & \langle \varnothing,\ ((\mathsf{t}^{\mathsf{int}} \to \mathsf{t}^{\mathsf{int}}, \xi_2) \to (\mathsf{t}^{\mathsf{int}} \to \mathsf{t}^{\mathsf{int}}, \xi_1), \xi_3),\ \{\{3\} \subseteq \xi_3, \{4\} \subseteq \xi_1\} \rangle \\
 & & \text{where } (\mathsf{int} \to \mathsf{int})^* = (\mathsf{t}^{\mathsf{int}} \to \mathsf{t}^{\mathsf{int}}, \xi_2)
\end{array}$$

Next, consider the function part, $\lambda^1 g^{(\mathsf{int}\to\mathsf{int})\to(\mathsf{int}\to\mathsf{int})}.\ g\ (g\ (\lambda^2 v^{\mathsf{int}}.\ v))$. We have:

$$\begin{array}{lcl}
\mathcal{I}(v) & = & \langle \{v : \mathsf{t}^{\mathsf{int}}\},\ \mathsf{t}^{\mathsf{int}},\ \varnothing \rangle \\
\mathcal{I}(\lambda^2 v.\ v) & = & \langle \varnothing,\ (\mathsf{t}^{\mathsf{int}} \to \mathsf{t}^{\mathsf{int}}, \xi_4),\ \{\{2\} \subseteq \xi_4\} \rangle \\
\mathcal{I}(g) & = & \langle \{g : \varphi'\},\ \varphi',\ \varnothing \rangle \text{where} \\
 & & \varphi' = ((\mathsf{int} \to \mathsf{int}) \to (\mathsf{int} \to \mathsf{int}))^* \\
 & & \quad = ((\mathsf{t}^{\mathsf{int}} \to \mathsf{t}^{\mathsf{int}}, \xi_5) \to (\mathsf{t}^{\mathsf{int}} \to \mathsf{t}^{\mathsf{int}}, \xi_6), \xi_7) \\
\mathcal{I}(g\ (\lambda^2 v.\ v)) & = & \langle g : \varphi',\ (\mathsf{t}^{\mathsf{int}} \to \mathsf{t}^{\mathsf{int}}, \xi_6),\ \{\{2\} \subseteq \xi_4, \xi_4 \subseteq \xi_5\} \rangle \text{since} \\
 & & \{(\mathsf{t}^{\mathsf{int}} \to \mathsf{t}^{\mathsf{int}}, \xi_4) \le (\mathsf{t}^{\mathsf{int}} \to \mathsf{t}^{\mathsf{int}}, \xi_5)\} \rightsquigarrow \{\xi_4 \subseteq \xi_5\} \\
\mathcal{I}(g) & = & \langle \{g : \varphi\},\ \varphi,\ \varnothing \rangle \text{where} \\
 & & \varphi = ((\mathsf{int} \to \mathsf{int}) \to (\mathsf{int} \to \mathsf{int}))^* \\
 & & \quad = ((\mathsf{t}^{\mathsf{int}} \to \mathsf{t}^{\mathsf{int}}, \xi_8) \to (\mathsf{t}^{\mathsf{int}} \to \mathsf{t}^{\mathsf{int}}, \xi_9), \xi_{10}) \\
\mathcal{I}(g\ (g\ (\lambda^2 v.\ v))) & = & \langle g : (\varphi \wedge \varphi'),\ (\mathsf{t}^{\mathsf{int}} \to \mathsf{t}^{\mathsf{int}}, \xi_9),\ \{\{2\} \subseteq \xi_4,\ \xi_4 \subseteq \xi_5,\ \xi_6 \subseteq \xi_8\} \rangle \text{since} \\
 & & \{(\mathsf{t}^{\mathsf{int}} \to \mathsf{t}^{\mathsf{int}}, \xi_6) \le (\mathsf{t}^{\mathsf{int}} \to \mathsf{t}^{\mathsf{int}}, \xi_8)\} \rightsquigarrow \{\xi_6 \subseteq \xi_8\}
\end{array}$$

Hence:

$$\mathcal{I}(\lambda^1 g.\ g\ (g\ (\lambda^2 v.\ v))) \quad = \quad \langle \varnothing,\ ((\varphi \wedge \varphi') \to (\mathsf{t}^{\mathsf{int}} \to \mathsf{t}^{\mathsf{int}}, \xi_9), \xi_{11}),\ C_1 \rangle$$

where $C_1 = \{\{2\} \subseteq \xi_4,\ \xi_4 \subseteq \xi_5,\ \xi_6 \subseteq \xi_8,\ \{1\} \subseteq \xi_{11}\}$.

Now to obtain the result for the entire expression T, we first rename the flow variables in the flow property for the argument twice (once for φ and once for φ') and obtain:

$$\{((\mathsf{t}^{\mathsf{int}} \to \mathsf{t}^{\mathsf{int}}, \xi_2') \to (\mathsf{t}^{\mathsf{int}} \to \mathsf{t}^{\mathsf{int}}, \xi_1'), \xi_3') \le ((\mathsf{t}^{\mathsf{int}} \to \mathsf{t}^{\mathsf{int}}, \xi_8) \to (\mathsf{t}^{\mathsf{int}} \to \mathsf{t}^{\mathsf{int}}, \xi_9), \xi_{10})\} \rightsquigarrow C_2$$

and

$$\{((\mathsf{t}^{\mathsf{int}} \to \mathsf{t}^{\mathsf{int}}, \xi_2'') \to (\mathsf{t}^{\mathsf{int}} \to \mathsf{t}^{\mathsf{int}}, \xi_1''), \xi_3'') \le ((\mathsf{t}^{\mathsf{int}} \to \mathsf{t}^{\mathsf{int}}, \xi_5) \to (\mathsf{t}^{\mathsf{int}} \to \mathsf{t}^{\mathsf{int}}, \xi_6), \xi_7)\} \rightsquigarrow C_3$$

where

$$C_2 = \{\xi_8 \subseteq \xi_2',\ \xi_1' \subseteq \xi_9,\ \xi_3' \subseteq \xi_{10}\} \text{ and } C_3 = \{\xi_5 \subseteq \xi_2'',\ \xi_1'' \subseteq \xi_6,\ \xi_3' \subseteq \xi_7\}$$

Finally:

$$\mathcal{I}(T) \quad = \quad \langle \varnothing,\ (\mathsf{t}^{\mathsf{int}} \to \mathsf{t}^{\mathsf{int}}, \xi_9),\ C \rangle$$

where $C = \{\{3\} \subseteq \xi_3', \{4\} \subseteq \xi_1', \{3\} \subseteq \xi_3'', \{4\} \subseteq \xi_1''\} \cup C_1 \cup C_2 \cup C_3$. The minimal solution for C, obtained by the transitive closure algorithm, is given by

$$[(\xi_{10}, \xi_3') \mapsto \{3\},\ (\xi_2'', \xi_5, \xi_4) \mapsto \{2\},\ (\xi_1', \xi_9) \mapsto \{4\},\ (\xi_2', \xi_8, \xi_6, \xi_1'') \mapsto \{4\},\ (\xi_7, \xi_3'') \mapsto \{3\}],$$

where $[(\xi, \xi') \mapsto L]$ abbreviates $[\xi \mapsto L,\ \xi' \mapsto L]$.

We can therefore conclude that the entire expression evaluates to λ^4 as we expect.

Moreover, note that the type of g in the body of λ^1 is $\varphi \wedge \varphi'$, where

$$\varphi = ((\mathsf{t}^{\mathsf{int}} \to \mathsf{t}^{\mathsf{int}}, \{4\}) \to (\mathsf{t}^{\mathsf{int}} \to \mathsf{t}^{\mathsf{int}}, \{4\}), \{3\})$$

and

$$\varphi' = ((\mathsf{t}^{\mathsf{int}} \to \mathsf{t}^{\mathsf{int}}, \{2\}) \to (\mathsf{t}^{\mathsf{int}} \to \mathsf{t}^{\mathsf{int}}, \{4\}), \{3\})$$

Thus λ^1 is applied to λ^3 and λ^3 exhibits the following behaviours: at one application site it obtains λ^2 as argument, yielding λ^4 as result. At another application site it obtains λ^4 as argument, yielding λ^4 as result. This conforms to our expectations in Section 2.

6.2. *Soundness of the inference algorithm*

Theorem 6.3 (Soundness)
Let $\mathcal{I}(e^\sigma) = \langle \pi,\ \varphi,\ C \rangle$. If S is a solution to C then $S(\pi) \vdash e : S(\varphi)$ is derivable in the property system.

Proof. By induction on the structure of e^σ.

- Case of x^σ: Let $\varphi = \sigma^*$. Then $\mathcal{I}(x^\sigma) = \langle \{x : \varphi\},\ \varphi,\ \varnothing \rangle$. Let S be any solution chosen for $\varnothing$. By rule **Var** in Table 2, $\{x : S(\varphi)\} \vdash x : S(\varphi)$.

- Case of n^{int}: Then $\mathcal{I}(n^{\mathsf{int}}) = \langle \varnothing,\ \mathsf{t}^{\mathsf{int}},\ \varnothing \rangle$. Let S be any solution chosen for $\varnothing$. By rule **Int** in Table 2, and since $S(\mathsf{t}^{\mathsf{int}}) = \mathsf{t}^{\mathsf{int}}$ for any S, we have, $\varnothing \vdash n^{\mathsf{int}} : \mathsf{t}^{\mathsf{int}}$.

- Case of $\lambda^\ell x^\sigma.\ e^\tau$: We have two cases: **Case (a)** $x \in \mathit{Dom}(\pi)$: Then $\mathcal{I}(\lambda^\ell x^\sigma.\ e^\tau) = \langle \pi',\ ((\bigwedge_{i\in I} \varphi_i) \to \varphi, \xi),\ C \cup \{\{\ell\} \subseteq \xi\} \rangle$, where $\mathcal{I}(e^\tau) = \langle \pi,\ \varphi,\ C \rangle$, and $\pi = \pi' \oplus \{x : \bigwedge_{i\in I} \varphi_i\}$. Let S be a solution to $C \cup \{\{\ell\} \subseteq \xi\}$. Then S is a solution to C, and, $\ell \in S(\xi)$; hence, by the induction hypothesis on e^τ, we have, $S(\pi) \vdash e : S(\varphi)$. Thus $S(\pi') \oplus \{x : \bigwedge_{i\in I} S(\varphi_i)\} \vdash e^\tau : S(\varphi)$. Therefore, by rule **Lam** in Table 2, $S(\pi') \vdash \lambda^\ell x^\sigma.\ e^\tau : S((\bigwedge_{i\in I} \varphi_i) \to \varphi, \xi)$. **Case (b)** $x \notin \mathit{Dom}(\pi)$: Then $\mathcal{I}(\lambda^\ell x^\sigma.\ e^\tau) =$
$\langle \pi,\ (\psi \to \varphi, \xi),\ C \cup \{\{\ell\} \subseteq \xi\} \rangle$, where $\psi = \sigma^*$ and $\mathcal{I}(e^\tau) = \langle \pi,\ \varphi,\ C \rangle$. Let S be a solution to $C \cup \{\{\ell\} \subseteq \xi\}$. Then S is a solution to C, and, $\ell \in S(\xi)$; hence, by the induction hypothesis on e^τ, we have, $S(\pi) \vdash e^\tau : S(\varphi)$; by weakening the property environment π, we have by Proposition 4.8, $S(\pi) \oplus \{x : \psi\} \vdash e^\tau : S(\varphi)$. Therefore, by rule **Lam** in Table 2, $S(\pi) \vdash \lambda^\ell x^\sigma.\ e^\tau : (\psi \to S(\varphi), S(\xi))$. Now since $\mathit{Dom}(S)$ is disjoint from set of label variables occurring in ψ, we get, $S(\pi) \vdash \lambda^\ell x^\sigma.\ e^\tau : S(\psi \to \varphi, \xi)$.

- Case of $e_1^{\sigma\to\tau} e_2^\sigma$: Then $\mathcal{I}(e_1^{\sigma\to\tau} e_2^\sigma) = \big\langle \pi + \sum_{i\in I} \widetilde{\pi}_i,\ \varphi_{12},\ C \big\rangle$, where $C = C_1 \cup (\bigcup_{i\in I} \widetilde{C_i}) \cup C'$. Let S be a solution to C. Then S is a solution to C_1, to $\widetilde{C_i}$, for $i \in I$ and to C'. Since $\mathcal{I}(e_1^{\sigma\to\tau}) = \langle \pi_1,\ \varphi_1,\ C_1 \rangle$, therefore, by the induction hypothesis on $e_1^{\sigma\to\tau}$, $\quad S(\pi_1) \vdash e_1^{\sigma\to\tau} : ((\bigwedge_{i\in I} S(\varphi_{1i})) \to S(\varphi_{12}), S(\kappa))$. Furthermore, since $\mathcal{I}(e_2^\sigma) = \langle \pi_2,\ \varphi_2,\ C_2 \rangle$, therefore, by the induction hypothesis on e_2^σ, $S(\pi_2) \vdash e_2^\sigma : S(\varphi_2)$, and, renaming for each $i \in I$ yields i disjoint triples $\big\langle \widetilde{\pi}_i,\ \widetilde{\varphi}_i,\ \widetilde{C_i} \big\rangle$ such that $S(\widetilde{\pi}_i) \vdash e_2^\sigma : S(\widetilde{\varphi}_i)$. Now since $\{\widetilde{\varphi}_i \le \varphi_{1i} \mid i \in I\} \overset{+}{\rightsquigarrow} C'$, and since S is a solution to C', therefore, by Lemma 6.2, $S(\widetilde{\varphi}_i) \le S(\varphi_{1i})$ for $i \in I$. By weakening of property environment $S(\pi_1)$, we obtain, $S(\pi_1) + \sum_{i\in I} S(\widetilde{\pi}_i) \vdash e_1^{\sigma\to\tau} : ((\bigwedge_{i\in I} S(\varphi_{1i})) \to S(\varphi_{12}), S(\kappa))$. Similarly, by weakening of property environment $S(\widetilde{\pi}_i)$, for each $i \in I$, we obtain, $S(\pi_1) + \sum_{i\in I} S(\widetilde{\pi}_i) \vdash e_2^\sigma : S(\widetilde{\varphi}_i)$.

Hence by rule **App** in Table 2, we obtain, $S(\pi_1 + \sum_{i\in I} \widetilde{\pi}_i) \vdash e_1^{\sigma\to\tau} e_2^{\sigma} : S(\varphi_{12})$.

- Case of $\mathsf{if}\ e_1^{\mathsf{int}}\ \mathsf{then}\ e_2^{\sigma}\ \mathsf{else}\ e_3^{\sigma}$: Then $\mathcal{I}(\mathsf{if}\ e_1^{\mathsf{int}}\ \mathsf{then}\ e_2^{\sigma}\ \mathsf{else}\ e_3^{\sigma}) = \langle \pi_1 + \pi_2 + \pi_3,\ \varphi,\ C\rangle$, where $C = C_1 \cup C_2 \cup C_3 \cup C_4$. Let S be a solution to C. Then S is a solution to C_1, C_2, C_3 and C_4. Since $\mathcal{I}(e_1^{\mathsf{int}}) = \langle \pi_1,\ \mathsf{t}^{\mathsf{int}},\ C_1\rangle$, therefore, by the induction hypothesis on e_1^{int}, $S(\pi_1) \vdash e_1^{\mathsf{int}} : \mathsf{t}^{\mathsf{int}}$. Since $\mathcal{I}(e_2^{\sigma}) = \langle \pi_2,\ \varphi_2,\ C_2\rangle$, therefore, by the induction hypothesis on e_2^{σ}, $S(\pi_2) \vdash e_2^{\sigma} : S(\varphi_2)$. Finally, since $\mathcal{I}(e_3^{\sigma}) = \langle \pi_3,\ \varphi_3,\ C_3\rangle$, therefore, by the induction hypothesis on e_3^{σ}, $S(\pi_3) \vdash e_3^{\sigma} : S(\varphi_3)$. Let $\sigma^* = \varphi$. Since S is a solution to C_4, and $\{\varphi_2 \le \varphi, \varphi_3 \le \varphi\} \overset{+}{\rightsquigarrow} C_4$, by Lemma 6.2, $S(\varphi_2) \le S(\varphi)$ and $S(\varphi_3) \le S(\varphi)$. Thus, by weakening of property environments and rule **If** in Table 2, $S(\pi_1 + \pi_2 + \pi_3) \vdash \mathsf{if}\ e_1^{\mathsf{int}}\ \mathsf{then}\ e_2^{\sigma}\ \mathsf{else}\ e_3^{\sigma} : S(\varphi)$.

- Case of $\mathsf{fun}^{\ell}\, f^{\sigma\to\tau}\,(x^{\sigma}) = e^{\tau}$: **Case(a)**: $x \notin FV(e)$: Then $\mathcal{I}(\mathsf{fun}^{\ell}\, f^{\sigma\to\tau}\,(x^{\sigma}) = e^{\tau}) = \langle \pi,\ (\sigma^* \to \delta, \xi),\ C \cup C_1 \cup \{\{\ell\} \subseteq \xi\}\rangle$. Let S be a solution to $C \cup C_1 \cup \{\{\ell\} \subseteq \xi\}$. Then S is a solution to C, C_1, and $\ell \in S(\xi)$. Let $\{(\sigma^* \to \delta, \xi) \le \bigwedge_{i\in I}(\varphi_i \to \varphi_i', \xi_i)\} \overset{+}{\rightsquigarrow} C_1$. Since S is a solution to C_1, by Lemma 6.2, $S(\sigma^* \to \delta, \xi) \le \bigwedge_{i\in I} S(\varphi_i \to \varphi_i', \xi_i)$. Since $\mathcal{I}(e^{\tau}) = \langle \pi \oplus \{f : \bigwedge_{i\in I}(\varphi_i \to \varphi_i', \xi_i)\},\ \delta,\ C\rangle$, therefore, by the induction hypothesis on e^{τ}, $S(\pi) \oplus \{f : \bigwedge_{i\in I} S(\varphi_i \to \varphi_i', \xi_i)\} \vdash e^{\tau} : S(\delta)$. By weakening, (Proposition 4.8), we obtain, $S(\pi) \oplus \{f : \bigwedge_{i\in I} S(\varphi_i \to \varphi_i', \xi_i), x : S(\sigma^*)\} \vdash e^{\tau} : S(\delta)$. Hence by rule **Fun** in Table 2, $S(\pi) \vdash \mathsf{fun}^{\ell}\, f^{\sigma\to\tau}\,(x^{\sigma}) = e^{\tau} : S((\bigwedge_{j\in J} \delta_j) \to \delta, \xi)$. **Case(b)**: $x \in FV(e)$: Then $\mathcal{I}(\mathsf{fun}^{\ell}\, f^{\sigma\to\tau}\,(x^{\sigma}) = e^{\tau}) = \left\langle \pi,\ ((\bigwedge_{j\in J} \delta_j) \to \delta, \xi),\ C \cup C_1 \cup \{\{\ell\} \subseteq \xi\}\right\rangle$. Let S be a solution to $C \cup C_1 \cup \{\{\ell\} \subseteq \xi\}$. Then S is a solution to C, C_1 and $\ell \in S(\xi)$. Since $\mathcal{I}(e^{\tau}) = \left\langle \pi \oplus \{f : \bigwedge_{i\in I}(\varphi_i \to \varphi_i', \xi_i), x : \bigwedge_{j\in J} \delta_j\},\ \delta,\ C\right\rangle$, therefore, by the induction hypothesis on e^{τ}, $S(\pi) \oplus \{f : \bigwedge_{i\in I} S(\varphi_i \to \varphi_i', \xi_i), x : \bigwedge_{j\in J} S(\delta_j)\} \vdash e^{\tau} : S(\delta)$. Let $\{((\bigwedge_{j\in J} \delta_j) \to \delta, \xi) \le \bigwedge_{i\in I}(\varphi_i \to \varphi_i', \xi_i)\} \overset{+}{\rightsquigarrow} C_1$. Since S is a solution to C_1, by Lemma 6.2, $S((\bigwedge_{j\in J} \delta_j) \to \delta, \xi) \le \bigwedge_{i\in I} S(\varphi_i \to \varphi_i', \xi_i)$. Hence by rule **Fun** in Table 2, $S(\pi) \vdash \mathsf{fun}^{\ell}\, f^{\sigma\to\tau}\,(x^{\sigma}) = e^{\tau} : S((\bigwedge_{j\in J} \delta_j) \to \delta, \xi)$.

- Case of $\mathsf{op}\ e^{\mathsf{int}}$: Then $\mathcal{I}(\mathsf{op}\ e^{\mathsf{int}}) = \langle \pi,\ \mathsf{t}^{\mathsf{int}},\ C\rangle$. Let S be a solution to C. Therefore, by the induction hypothesis on e^{int}, $S(\pi) \vdash e^{\mathsf{int}} : \mathsf{t}^{\mathsf{int}}$. Thus using rule **Op** in Table 2, $S(\pi) \vdash \mathsf{op}\ e^{\mathsf{int}} : \mathsf{t}^{\mathsf{int}}$. □

Now we can revisit the type soundness theorem, Theorem 5.3. The main import of the theorem is in conjunction with the soundness theorem for $\mathcal{I}$: suppose $S(\pi) \vdash e : S(\varphi)$ as in Theorem 6.3. Let $e \to e'$ as in Theorem 5.3. Then there exists φ', such that $S(\pi) \vdash e' : \varphi'$ and $\varphi' \le S(\varphi)$. In particular, if e is of function type, by the ordering on properties, the outermost set of labels associated with φ' will be contained in the outermost set of labels associated with $S(\varphi)$. That is, the evaluation of e yields a function which is predicted by the analysis. Hence the analysis is sound.

6.3. *Completeness of the inference algorithm*

The inference algorithm $\mathcal{I}$ defined in Table 4 is complete. This means that for any expression e, if $\mathcal{I}(e) = \langle \pi,\ \varphi,\ C\rangle$ then $\pi \vdash e : \varphi$ is a principal typing for e. Any

other typing for e can be obtained as an instance of the above typing. This is formalised in the following completeness theorem.

Theorem 6.4 (Completeness)
Let $\mathcal{I}(e) = \langle \pi,\ \varphi,\ C\rangle$. Suppose $\eta \vdash e^{\sigma} : \psi$ is derivable and that η is a rank 1 property environment, ψ is a rank 2 property and η, ψ are ground. Then there exists a substitution S, such that:

(i) S is a solution to C.
(ii) $S\langle \pi,\ \varphi\rangle \leq \langle \eta,\ \psi\rangle$, *i.e.*, $\eta \leq S(\pi)$ and $S(\varphi) \leq \psi$.

Proof. • Case of x^{σ}: The derivation is: $\eta \oplus \{x : \bigwedge_{i\in I} \psi_i\} \vdash x^{\sigma} : \psi_j$, where $j \in I$. By the inference algorithm, $\mathcal{I}(x^{\sigma}) = \langle \{x : \varphi\},\ \varphi,\ \varnothing\rangle$, where $\varphi = \sigma^*$. We need a substitution, S, such that, S is a solution to $\varnothing$ (holds trivially for any S) and $S(\langle \{x : \varphi\},\ \varphi\rangle) \leq \langle \eta \oplus \{x : \bigwedge_{i\in I} \psi_i\},\ \psi_j\rangle$. That is:

$$\textbf{(Ai)}\ \ \eta \oplus \{x : \textstyle\bigwedge_{i\in I} \psi_i\} \leq S(\{x : \varphi\}) \quad \textbf{(Aii)}\ \ S(\varphi) \leq \psi_j.$$

To prove **(Aii)**, note that $\psi_j \in \mathit{Prop}_0(\sigma)$. Hence, by Proposition 4.2, there exists a substitution, S', such that $S'(\sigma^*) = \psi_j$. But $\varphi = \sigma^*$. Therefore, choose S to be S', so that $S'(\varphi) = S'(\sigma^*) = \psi_j$. Thus $S'(\varphi) \leq \psi_j$.
To prove **(Ai)**, we need only show that for all $x \in \mathit{Dom}(\{x : \varphi\})$, it is the case that $x \in \mathit{Dom}(\eta \oplus \{x : \bigwedge_{i\in I} \psi_i\})$, and $\bigwedge_{i\in I} \psi_i \leq S'(\varphi)$, *i.e.*, $\bigwedge_{i\in I} \psi_i \leq \psi_j$. But this holds by the ordering on properties (Section 4) since $j \in I$.

• Case of n^{int}: The derivation is: $\eta \vdash n^{\mathsf{int}} : \mathtt{t}^{\mathsf{int}}$. By the inference algorithm, $\mathcal{I}(n^{\mathsf{int}}) = \langle \varnothing,\ \mathtt{t}^{\mathsf{int}},\ \varnothing\rangle$. Case (i) of the theorem holds trivially for any substitution S. For case (ii), we need a substitution, S, such that $S(\mathtt{t}^{\mathsf{int}}) \leq \mathtt{t}^{\mathsf{int}}$. Choose S to be empty, *i.e.*, $[\,]$.

• Case of $\lambda^{\ell} x^{\sigma}.\ e^{\tau}$: The derivation is:

$$\frac{\eta' \oplus \{x : \bigwedge_{j\in J} \psi_j\} \vdash e^{\tau} : \psi \quad \ell \in L}{\eta' \vdash \lambda^{\ell} x^{\sigma}.\ e^{\tau} : ((\bigwedge_{j\in J} \psi_j) \to \psi, L)}\ \eta = \eta' \oplus \{x : \bigwedge_{j\in J} \psi_j\}$$

By the inference algorithm, we have two cases. **Case(a):** $\mathcal{I}(\lambda^{\ell} x^{\sigma}.\ e^{\tau}) =$
let $\langle \pi,\ \varphi,\ C\rangle = \mathcal{I}(e^{\tau})$ **in** $\langle \pi',\ ((\bigwedge_{i\in I} \varphi_i) \to \varphi, \xi),\ C \cup \{\{\ell\} \subseteq \xi\}\rangle$ where $\pi = \pi' \oplus \{x^{\sigma} : \bigwedge_{i\in I} \varphi_i\}$. By the induction hypothesis on e^{τ}, there exists a substitution, S, such that:

$$\textbf{(Ai)}\ S \text{ is a solution for } C \qquad \textbf{(Aii)}\ \eta \leq S(\pi) \qquad \textbf{(Aiii)}\ S(\varphi) \leq \psi$$

From **A(ii)**, we obtain, in particular:

$$\textbf{(Bi)}\ \ \textstyle\bigwedge_{j\in J}(\psi_j) \leq \bigwedge_{i\in I} S(\varphi_i) \qquad \textbf{(Bii)}\ \ \eta' \leq S(\pi')$$

To prove the theorem, we need a substitution, S', such that, **(Ci)** S' is a solution for $C \cup \{\{\ell\} \subseteq \xi\}$,

$$\textbf{(Cii)}\ \ \eta' \leq S'(\pi') \qquad \textbf{(Ciii)}\ \ S'((\textstyle\bigwedge_{i\in I} \varphi_i) \to \varphi, \xi) \leq ((\bigwedge_{j\in J} \psi_j) \to \psi, L).$$

Since fresh ξ, and S is a solution for C, and $\ell \in L$, therefore, choose the solution for **(Ci)** as $S' = [\xi \mapsto \{\ell\}] \circ S$. Then **(Cii)** follows from **(Bii)**, and, **(Ciii)** follows from **(Bi)**, **(Aiii)** and $\ell \in L$.

Case(b): $\mathcal{I}(\lambda^{\ell} x^{\sigma}.\ e^{\tau}) = \textbf{let } \langle \pi,\ \varphi,\ C\rangle = \mathcal{I}(e^{\tau}) \textbf{ in } \langle \pi,\ (\varphi' \to \varphi, \xi),\ C \cup \{\{\ell\} \subseteq \xi\}\rangle$ where $\varphi' = \sigma^*$ *and* $x \notin \mathit{Dom}(\pi)$. By the induction hypothesis on e^{τ}, there exists a substitution, S, such that:

$$\textbf{(Ai)}\ S \text{ is a solution for } C \qquad \textbf{(Aii)}\ \eta \leq S(\pi) \qquad \textbf{(Aiii)}\ S(\varphi) \leq \psi$$

To prove the theorem, we need a substitution, S', such that, **(Ci)** S' is a solution for $(C \cup \{\{\ell\} \subseteq \xi\})$,

$$\textbf{(Cii)}\ \eta' \leq S'(\pi'), \qquad \textbf{(Ciii)}\ S'(\varphi' \to \varphi, \xi) \leq ((\textstyle\bigwedge_{j\in J} \psi_j) \to \psi, L).$$

From **(Ciii)**, this means we need S' such that:

$$\textbf{(Civ)}\ \bigwedge_{j\in J}(\psi_j) \leq S'(\varphi') \quad \textbf{(Cv)}\ S'(\varphi) \leq \psi \quad \textbf{(Cvi)}\ S'(\xi) \subseteq L$$

To prove **C(iv)**, note that each $\psi_j \in \mathit{Prop}_0(\sigma)$. Since $\varphi' = \sigma^*$, there exists $\widehat{j} \in J$ and substitution $\widehat{S}$ such that, by Proposition 4.2, $\widehat{S}(\sigma^*) = \psi_{\widehat{j}}$. Thus, $\widehat{S}(\varphi') = \psi_{\widehat{j}}$. Therefore, by the ordering on properties, $\bigwedge_{j\in J}(\psi_j) \leq \widehat{S}(\varphi')$. Since fresh ξ and S is a solution to C, therefore, choose the solution to **(Ci)** as $S' = [\xi \mapsto \{\ell\}] \circ \widehat{S} \circ S$. Then it is easy to see that **(Cii)** follows from **(Aii)**, **(Cv)** follows from **(Aiii)**, and **(Cvi)** follows since $\ell \in L$.

- Case of $e_1^{\sigma\to\tau} e_2^{\sigma}$: The derivation is:

$$\frac{\eta \vdash e_1^{\sigma\to\tau} : ((\textstyle\bigwedge_{i\in I} \psi_i) \to \psi, \kappa) \quad \forall i \in I :\ \eta \vdash e_2^{\sigma} : \psi_i' \quad \forall i \in I :\ \psi_i' \leq \psi_i}{\eta \vdash e_1^{\sigma\to\tau} e_2^{\sigma} : \psi}$$

By the inference algorithm, we have,

$$\begin{array}{rcl}
\mathcal{I}(e_1^{\sigma\to\tau} e_2^{\sigma}) & = & \textbf{let } \langle \pi_1,\ \varphi_1,\ C_1\rangle = \mathcal{I}(e_1^{\sigma\to\tau}) \\
& & \quad \langle \pi_2,\ \varphi_2,\ C_2\rangle = \mathcal{I}(e_2^{\sigma}) \\
& & \textbf{in case } \varphi_1 \textbf{ of} \\
& & \quad ((\bigwedge_{j\in J} \varphi_{1j}) \to \varphi_{12}, \mu) : \\
& & \qquad \textbf{let } \left\langle \widetilde{\pi_j},\ \widetilde{\varphi_j},\ \widetilde{C_j} \right\rangle = \mathit{Rename}(\pi_2, \varphi_2, C_2) \\
& & \qquad \{\widetilde{\varphi_j} \leq \varphi_{1j} \mid j \in J\} \overset{+}{\rightsquigarrow} \widehat{C} \\
& & \qquad C' = C_1 \cup (\bigcup_{j\in J} \widetilde{C_j}) \cup \widehat{C} \\
& & \qquad \textbf{in } \left\langle \pi_1 + \textstyle\sum_{j\in J} \widetilde{\pi_j},\ \varphi_{12},\ C' \right\rangle
\end{array}$$

By the induction hypothesis on $e_1^{\sigma\to\tau}$, there exists substitution, S_1, such that **(Ai)** S_1 is a solution for C_1,

$$\textbf{(Aii)}\ \eta \leq S_1(\pi_1)\ \ \textbf{(Aiii)}\ S_1((\bigwedge_{j\in J} \varphi_{1j}) \to \varphi_{12}, \mu) \leq ((\bigwedge_{i\in I} \psi_i) \to \psi, \kappa)$$

From **(Aiii)**, we know that:

$$\textbf{(Aiv)}\ \ \textstyle\bigwedge_{i\in I}(\psi_i) \le \bigwedge_{j\in J} S_1(\varphi_{1j}) \qquad \textbf{(Av)}\ \ S_1(\varphi_{12}) \le \psi \qquad \textbf{(Avi)}\ \ S_1(\mu) \subseteq \kappa.$$

From **(Aiv)**, it must be the case that, for all $j \in J$, there exists $i_j \in I$, such that:

$$\textbf{(Avii)}\ \ \psi_{i_j} \le S_1(\varphi_{1j})$$

By the induction hypothesis on e_2^σ, for all $j \in J$, there exists substitution, $\widetilde{S_j}$ and there exists $i \in I$ such that:

$$\textbf{(Bi)}\ \ S_j \text{ is a solution for } \widetilde{C_j} \qquad \textbf{(Bii)}\ \ \eta \le \widetilde{S_j}(\widetilde{\pi_j}) \qquad \textbf{(Biii)}\ \ \widetilde{S_j}(\widetilde{\varphi_j}) \le \psi'_{i_j}.$$

To prove the theorem, we need a substitution S', such that:

$$\textbf{(Ci)}\ \ S' \text{ is a solution for } C' \qquad \textbf{(Cii)}\ \ \eta \le S'(\pi_1 + \textstyle\sum_{j\in J} \widetilde{\pi_j}) \qquad \textbf{(Ciii)}\ \ S'(\varphi_{12}) \le \psi.$$

Choose $S' = S_1 \circ \bigcirc_{j\in J}\widetilde{S_j}$. Then **(Cii)** holds by **(Aii)**, **(Bii)** and Proposition 4.7, and, **(Ciii)** holds by **(Av)**. To show **(Ci)**, we need to establish that S' is a solution for $\widehat{C}$; for this, the crucial bit is to show that $S'(\widetilde{\varphi_j}) \le S'(\varphi_{1j})$, for every $j \in J$; this we obtain as follows:

$$S'(\widetilde{\varphi_j}) \le \psi'_{i_j}\ \text{(by \textbf{Biii})}; \quad \psi'_{i_j} \le \psi_{i_j}\ \text{(by hypothesis in derivation)};$$
$$\psi_{i_j} \le S'(\varphi_{1j})\ \text{(by \textbf{Avii})}$$

Therefore, $S'(\widetilde{\varphi_j}) \le S'(\varphi_{1j})$ for every $j \in J$; hence S' is a solution for $\widehat{C}$ by Lemma 6.2. By **(Ai)**, S' is a solution for C_1, and by **(Bi)**, S' is a solution for $\widetilde{C_j}$, for all $j \in J$.

- Case of $\mathsf{if}\ e_1^{\mathsf{int}}\ \mathsf{then}\ e_2^\sigma\ \mathsf{else}\ e_3^\sigma$: The derivation is:

$$\frac{\eta \vdash e_1^{\mathsf{int}} : \mathtt{t}^{\mathsf{int}} \quad \eta \vdash e_2^\sigma : \psi_2 \quad \eta \vdash e_3^\sigma : \psi_3 \quad \psi_2 \le \psi \quad \psi_3 \le \psi}{\eta \vdash \mathsf{if}\ e_1^{\mathsf{int}}\ \mathsf{then}\ e_2^\sigma\ \mathsf{else}\ e_3^\sigma : \psi}$$

By the inference algorithm,

$$\begin{array}{l}\mathcal{I}(\mathsf{if}\ e_1^{\mathsf{int}}\ \mathsf{then}\ e_2^\sigma\ \mathsf{else}\ e_3^\sigma) = \\ \quad \textbf{let } \langle \pi_1,\ \mathtt{t}^{\mathsf{int}},\ C_1\rangle = \mathcal{I}(e_1^{\mathsf{int}}) \\ \qquad \langle \pi_2,\ \varphi_2,\ C_2\rangle = \mathcal{I}(e_2^\sigma) \\ \qquad \langle \pi_3,\ \varphi_3,\ C_3\rangle = \mathcal{I}(e_3^\sigma) \\ \qquad \varphi = \sigma^* \\ \qquad \{\varphi_2 \le \varphi, \varphi_3 \le \varphi\} \overset{+}{\rightsquigarrow} C_4 \\ \qquad C = C_1 \cup C_2 \cup C_3 \cup C_4 \\ \quad \textbf{in } \langle \pi_1 + \pi_2 + \pi_3,\ \varphi,\ C\rangle\end{array}$$

By the induction hypothesis on $e_1^{\mathsf{int}},\ e_2^\sigma,\ e_3^\sigma$, there exist substitutions $S_1,\ S_2,\ S_3$, such that

(Ai) S_1 is a solution for C_1	**(Aii)** $\eta \le S_1(\pi_1)$	**(Aiii)** $S_1(\mathtt{t}^{\mathsf{int}}) \le \mathtt{t}^{\mathsf{int}}$
(Bi) S_2 is a solution for C_2	**(Bii)** $\eta \le S_2(\pi_2)$	**(Biii)** $S_2(\varphi_2) \le \psi_2$
(Ci) S_3 is a solution for C_3	**(Cii)** $\eta \le S_3(\pi_3)$	**(Ciii)** $S_3(\varphi_3) \le \psi_3$

Let $\pi = \pi_1 + \pi_2 + \pi_3$. To prove the theorem, we need substitution S, such that

$$\textbf{(Di)}\ \ S \text{ is a solution for } C \qquad \textbf{(Dii)}\ \ \eta \leq S(\pi) \qquad \textbf{(Diii)}\ \ S(\varphi) \leq \psi$$

By Proposition 4.4, there exists a substitution, S_4, such that $S_4(\sigma^*) \leq \psi$. Since $\varphi = \sigma^*$, we obtain, $S_4(\varphi) \leq \psi$. Now choose the substitution, S, to be $S_4 \circ S_3 \circ S_2 \circ S_1$. Hence **(Diii)** holds. Clearly, **(Di)** holds by **(Ai), (Bi), (Ci)** and **(Dii)** holds by **(Aii), (Bii), (Cii)** and Proposition 4.7.

- Case of $\mathsf{fun}^{\ell}\, f^{\sigma\to\tau}\,(x^{\sigma}) = e^{\tau}$: The derivation is:

$$\frac{\begin{array}{l}\eta \oplus \{f : \bigwedge_{i\in I}(\psi_i \to \psi'_i, L_i), x : \bigwedge_{j\in J}\theta_j\} \vdash e^{\tau} : \theta \\ \forall i \in I : ((\bigwedge_{j\in J}\theta_j) \to \theta, L) \leq (\psi_i \to \psi'_i, L_i) \\ \ell \in L\end{array}}{\eta \vdash \mathsf{fun}^{\ell}\, f^{\sigma\to\tau}\,(x^{\sigma}) = e^{\tau} : ((\bigwedge_{j\in J}\theta_j) \to \theta, L)}$$

By the inference algorithm, we have two cases, of which we will just prove the case when $x \in FV(e)$. The other case is similar. Let $\eta' = \eta \oplus \{f : \bigwedge_{i\in I}(\psi_i \to \psi'_i, L_i), x : \bigwedge_{j\in J}\theta_j\}$.

$$\begin{array}{rcl}\mathcal{I}(\mathsf{fun}^{\ell}\, f^{\sigma\to\tau}\,(x^{\sigma}) = e^{\tau}) & = & \textbf{let} \\ & & \quad \langle \pi', \delta, C\rangle = \mathcal{I}(e^{\tau}) \\ & & \quad \{((\bigwedge_{m\in M}\delta_m) \to \delta, \xi) \leq \bigwedge_{k\in K}(\varphi_k \to \varphi'_k, \xi_k)\} \overset{+}{\rightsquigarrow} C_1 \\ & & \textbf{in}\ \langle \pi,\ \ ((\bigwedge_{m\in M}\delta_m) \to \delta,\ \xi),\ \ C \cup C_1 \cup \{\{\ell\} \subseteq \xi\}\rangle\end{array}$$

where $\pi' = \pi \oplus \{f : \bigwedge_{k\in K}(\varphi_k \to \varphi'_k, \xi_k), x : \bigwedge_{m\in M}\delta_m\}$. By the induction hypothesis on e^{τ}, there exists substitution, S, such that

$$\textbf{(Ai)}\ \ S \text{ is a solution for } C \qquad \textbf{(Aii)}\ \ \eta' \leq S(\pi') \qquad \textbf{(Aiii)}\ \ S(\delta) \leq \theta$$

From **(Aii)** we obtain: **(Bi)** $\eta \leq S(\pi)$,

$$\textbf{(Bii)}\ \ \bigwedge_{i\in I}(\psi_i \to \psi'_i, L_i) \leq \bigwedge_{k\in K} S(\varphi_k \to \varphi'_k, \xi_k) \qquad \textbf{(Biii)}\ \ \bigwedge_{j\in J}\theta_j \leq S(\bigwedge_{m\in M}\delta_m)$$

From **(Bii)**, for all $k \in K$, there exists $i_k \in I$, such that, **(Biv)** $(\psi_{i_k} \to \psi'_{i_k}, L_{i_k}) \leq S(\varphi_k \to \varphi'_k, \xi_k)$. From **(Biv)**, we obtain, for all $k \in K$ and $i_k \in I$,

$$\textbf{(Bv)}\ \ S(\varphi_k) \leq \psi_{i_k} \qquad \textbf{(Bvi)}\ \ \psi'_{i_k} \leq S(\varphi'_k) \qquad \textbf{(Bvii)}\ \ L_{i_k} \subseteq S(\xi_k)$$

To prove the theorem, we need substitution, S', such that: **(Ci)** S' is a solution for $C \cup C_1 \cup \{\{\ell\} \subseteq \xi\})$,

$$\textbf{(Cii)}\ \ \eta \leq S'(\pi) \qquad \textbf{(Ciii)}\ \ S'((\bigwedge_{m\in M}\delta_m) \to \delta, \xi) \leq ((\bigwedge_{j\in J}\theta_j) \to \theta, L)$$

Note that ξ is fresh and choose substitution $S' = [\xi \mapsto \{\ell\}] \circ S$. (Hence $\xi \notin Dom(S)$). Now **(Cii)** holds by **(Bi)**. To show **(Ciii)**, we need to show:

$$\textbf{(Di)}\ \ \bigwedge_{j\in J}\theta_j \leq S'(\bigwedge_{m\in M}\delta_m) \qquad \textbf{(Dii)}\ \ S'(\delta) \leq \theta \qquad \textbf{(Diii)}\ \ S'(\xi) \subseteq L$$

But now, **(Di)** follows from **(Biii)**, **(Dii)** follows from **(Aiii)**, and, **(Diii)** follows since $\ell \in L$.

To show **(Ci)**, the crucial bit is to show that S' is a solution for C_1 since **(Ai)** shows that S' is a solution for C and since $S'(\xi) = \{\ell\} \subseteq L$ by the derivation. It is enough to show, for all $k \in K$, that S' satisfies $((\bigwedge_{m\in M} \delta_m) \to \delta,\ \xi) \leq (\varphi_k \to \varphi'_k,\ \xi_k)$, then by Lemma 6.2, assert that S' is a solution to C_1. Thus we need to show, for all $k \in K$,

(Ei) $S'(\varphi_k) \leq \bigwedge_{m\in M} S'(\delta_m)$ **(Eii)** $S'(\delta) \leq S'(\varphi'_k)$ **(Eiii)** $S'(\xi) \subseteq S'(\xi_k)$

To show **(Ei)**, first note that by **(Bv)**, $S'(\varphi_k) \leq \psi_{i_k}$. By the derivation, we know, for all $i \in I$, that $((\bigwedge_{j\in J} \theta_j) \to \theta, L) \leq (\psi_i \to \psi'_i, L_i)$. That is, by contravariance, $\psi_{i_k} \leq \bigwedge_{j\in J} \theta_j$. But, by **(Di)**, $\bigwedge_{j\in J} \theta_j \leq \bigwedge_{m\in M} S'(\delta_m)$. Hence, it follows that $S'(\varphi_k) \leq \bigwedge_{m\in M} S'(\delta_m)$.

To show **(Eii)**, note that by **(Aiii)**, $S'(\delta) \leq \theta$. By the derivation, we know, for all $i \in I$, that $((\bigwedge_{j\in J} \theta_j) \to \theta, L) \leq (\psi_i \to \psi'_i, L_i)$. That is, by covariance, $\theta \leq \psi'_{i_k}$. But, by **(Bvi)**, $\psi'_{i_k} \leq S'(\varphi'_k)$. Hence, it follows that $S'(\delta) \leq S'(\varphi'_k)$.

To show **(Eiii)**, we know by the derivation that for all $i \in I : ((\bigwedge_{j\in J} \theta_j) \to \theta, L) \leq (\psi_i \to \psi'_i, L_i)$. Hence for all $i \in I : L \subseteq L_i$, so in particular, $L \subseteq L_{i_k}$. Now by **(Bviii)**, $L \subseteq S'(\xi_k)$. But $S'(\xi) = \ell \in L$, therefore, $\{\ell\} \subseteq S'(\xi_k)$.

- Case of $\mathsf{op}\ e^{\mathsf{int}}$: Easy and omitted. □

7. Related work

(Jim 1995; Jim 1996) advocates the use of rank 2 intersection types for typing the lambda calculus with a recursion operator, building on earlier work on intersection type systems by among others (Leivant 1983; van Bakel 1992; Coppo and Giannini 1992). 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 type inference in 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 a substitution instance of (A, τ). 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. This is not the case for the constraints generated by our control flow analysis.

7.1. *Type-based program analysis*

(Kuo and Mishra 1989) propose a type inference for strictness analysis without conjunctions. This is extended to conjunctions by (Benton 1992) and (Jensen 1991) and to conjunctive and disjunctive strictness types by (Jensen 1997). None of them propose inference algorithms for the systems. (Hankin and Le Métayer 1994; Hankin and Le Métayer 1995) 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 1992). For a given term e *and* property φ, this abstract machine will 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 a method that infers a property for the program.

(Jensen 1998) has proposed a strictness analysis for higher-order functional languages based on intersection types and parametric polymorphism. This combination allows to write certain strictness types in a very compact way. For example, a binary function that is strict in each argument (*e.g.*, addition) is given the type

$$\forall \alpha_1, \alpha_2 \,.\, \alpha_1 \to \alpha_2 \to \alpha_1 \wedge \alpha_2.$$

By instantiating the α's appropriately, all strictness properties of such a function can be obtained. These types are used to define a modular inference algorithm in the style of this paper. For each expression, the algorithm infers a property and an environment of hypotheses on the free variables. The inference algorithm follows the same structure as in this paper. However, the constraints that result from inference are over a different set of properties that are axiomatised differently and hence requires another constraint resolution technique.

(Henglein and Mossin 1994) and (Dussart et al. 1995) 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. Judgments in their logic are of the form

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

where A is a set of conjunctions and C is 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 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.

7.2. *Type-based control flow analysis*

Control flow analysis for higher-order functional languages (also called closure analysis) originated with the work by (Jones 1981; Jones 1987), (Sestoft 1988; Sestoft 1991) and (Shivers 1991). Several authors have extended this work to frameworks for control-flow analysis that can be instantiated to yield different mono- or polyvariant whole-program analyses (Jagannathan and Weeks 1995; Schmidt 1995; Jagannathan et al. 1997; Nielson and Nielson 1997; Amtoft and Turbak 2000; Palsberg and Pavloupoulou 2001; Schmidt 1995). These frameworks are derived from an operational semantics of the language and thus provide the semantic foundations for control flow analysis but they do not address the problem of modularising the analysis.

The algorithmic aspects of control flow analysis is addressed by (Palsberg 1995) who shows how a control flow analysis can be reduced to solving set constraints. Palsberg and O'Keefe show that Amadio and Cardelli's type system for the untyped lambda calculus with recursive types and subtyping (Amadio and Cardelli 1993) is equivalent to *safety*

analysis (Palsberg and O'Keefe 1995; Palsberg and Schwartzbach 1995). Safety analysis for the untyped lambda calculus is a global program analysis that detects run-time errors due to illegal use of operators,*e.g.*, when a constant is applied to a function. The equivalence says that a term is declared safe by the analysis if and only if it is typable in Amadio and Cardelli's type system. (Heintze 1995) shows that given a variety of type systems instrumented by control-flow information there exist corresponding control-flow analysis augmented by type information such that each type system is equivalent to its corresponding control-flow analysis. Equivalence here means that each system calculates the same flow and type information. Both of the above papers use type systems for control-flow analysis, but their methods of computation of flow information differ: in Palsberg and O'Keefe's work, the information is indirectly obtained via a safety analysis, while for an expression in Heintze's system, it is obtained by enumerating all of the possible control-flow types of the expression, and then systematically removing the (structural) type information. The upshot is that both methods lead to *global analyses* of expressions, in a (type-based) setting where most analyses usually rely on compositional inference algorithms to calculate program properties.

In her Phd thesis, Tang provides a *type and effect discipline* for a call-tracking analysis of the simply-typed lambda calculus with a recursion operator (Tang 1994). The analysis computes what functions may be called at a program point. Types are annotated with *control-flow effects* that statically approximate the set of functions called during the evaluation of an expression. *Subtyping* is used to obtain better precision for this analysis, by disambiguating call contexts. Tang's framework is attractive because it is *local*: given an expression e and an annotated-type environment Γ containing the annotations of the free variables in e, the (ML-style) type inference algorithm can *locally analyse* proper subexpressions of e, independently of the rest of the program; subsequently, it can compose the local analyses to obtain an analysis for the entire expression. Tang's analysis *is* modular: though she only proves that the analysis has principal types, it is easy to show that it also has principal typings.

This article extends the results of an earlier paper (Banerjee 1997) that developed a modular and polyvariant control-flow analysis for untyped programs. In that paper, the properties inferred were rank 2 intersection properties and type inference and control-flow analysis were done in a single pass. The paper did not address control-flow analysis for recursive function definitions which the current paper does. The control-flow analysis proposed by Banerjee in the earlier paper has been used extensively by the Church Project (Church project) in the implementation of a flow-based compiler for Standard ML of New Jersey (Dimock et al. 1997; Dimock et al. 2001).

The control flow analysis most closely related to ours is that of (Mossin 1997a),(Mossin 1997b, Chapter 6) who presents an intersection type-based control-flow analysis. A reduced version of Mossin's analysis (product types are not considered) is shown in Table 5. Mossin is mainly interested in the theoretical aspects and proves that the analysis is "exact" in the sense that if it calculates the control flow under *all* reduction strategies (and not just *e.g.* innermost or outermost reduction). He does not consider the algorithmic aspects of implementing the analysis but mentions in passing that the problem it solves is non-elementary recursive. The difference between our system and that of Mossin is

Properties: For each type σ define the set of properties $L_{CF}(\sigma)$ as follows. Here, $\mathit{Labels}(\sigma)$ denotes the set of labels of terms of type σ.

$$\frac{M \subseteq \mathit{Labels}(\texttt{Bool})}{\mathrm{Bool}^M \in L_{CF}(\mathsf{Bool})}$$

$$\frac{\varphi_1 \in L_{CF}(\sigma_1) \quad \varphi_2 \in L_{CF}(\sigma_2) \quad M \subseteq \mathit{Labels}(\sigma_1 \to \sigma_2)}{\varphi_1 \xrightarrow{M} \varphi_2 \in L_{CF}(\sigma_1 \to \sigma_2)}$$

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

Axiomatisation: (I denotes a finite, non-empty set).

- $\dfrac{\varphi \leq \psi_i, \quad i \in I}{\varphi \leq \bigwedge_{i \in I} \psi_i}$
- $\bigwedge_{i \in I} \varphi_i \leq \varphi_i, \quad \forall i \in I$
- $\dfrac{\psi_1 \leq \varphi_1, \varphi_2 \leq \psi_2 \quad M \subseteq N}{\varphi_1 \xrightarrow{M} \varphi_2 \leq \psi_1 \xrightarrow{N} \psi_2}$
- $\varphi \xrightarrow{M} \psi_1 \wedge \varphi \xrightarrow{M} \psi_2 = \varphi \xrightarrow{M} \psi_1 \wedge \psi_2$

Inference rules:

$$\textbf{Var} \quad \Delta[x \mapsto \varphi] \vdash_{CF} x : \varphi$$

$$\textbf{Const} \quad \frac{c^l \in \{\texttt{true}, \texttt{false}\}}{\Delta \vdash_{CF} c : \mathrm{Bool}^{\{l\}}}$$

$$\textbf{Abs} \quad \frac{\Delta[x \mapsto \varphi] \vdash_{CF} e : \psi}{\Delta \vdash_{CF} \lambda x^l.e : \varphi \xrightarrow{\{l\}} \psi}$$

$$\textbf{Fix} \quad \frac{\forall i \in I. \Delta[x : \psi] \vdash_{CF} e : \psi_i \quad \bigwedge_{i \in I} \psi_i \leq \psi}{\Delta \vdash_{CF} (\mathsf{fix}^l x.e) : \psi_j} \text{for any } j \in I$$

$$\textbf{App} \quad \frac{\Delta \vdash_{CF} e_1 : (\varphi_1 \xrightarrow{M} \varphi_2) \quad \forall i \in I. \Delta \vdash_{CF} e_2 : \psi_i \quad \bigwedge_{i \in I} \psi_i \leq \varphi_1}{\Delta \vdash_{CF} e_1 e_2 : \varphi_2}$$

$$\textbf{If} \quad \frac{\Delta \vdash_{CF} b : \mathrm{Bool}^M \quad \Delta \vdash_{CF} e_1 : \varphi_1 \quad \Delta \vdash_{CF} e_2 : \varphi_2 \quad \varphi_1 \leq \varphi \quad \varphi_2 \leq \varphi}{\Delta \vdash_{CF} \mathsf{if}\ b\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \varphi}$$

Table 5. *Mossin's control-flow logic (without products)*

that we have imposed a restriction that the function properties appearing in the inference rules must be of rank 2. This is done in order that the constraints in the inference algorithm have a simple form.

Finally, as discussed by (Amtoft and Turbak 2000; Palsberg and Pavloupoulou 2001), a more general system of polyvariant control-flow analysis may include both intersection and union types.

8. Discussion

We have developed a modular and polyvariant control-flow analysis for simply-typed program fragments. The inherent polymorphism of intersection types is exploited to provide a polyvariant analysis. The analysis is performed by a sound and complete inference algorithm that infers rank 2 intersection properties. The algorithm is compositional and works in a bottom-up manner inferring property environments as well as control-flow properties of program fragments. The completeness result shows that the algorithm computes principal typings: this facilitates linking of program fragments without reanalysis.

The inference algorithm for the application, $e_1 e_2$, in Table 4, reveals that the merging of the type environments in the result does *not* unify the possibly different types of a free variable occurring both in e_1 and in e_2: rather the variable is given an intersection type. This is of crucial importance in providing polyvariance.

The assumption of a simply-typed base language is exploited in the inference algorithm in computing properties of variables. If a variable has type σ, its control-flow property must have the same shape. For instance if $\sigma = \mathsf{int} \to \mathsf{int}$, then any function that flows to x must have the property $\sigma^* = (\mathtt{t}^{\mathsf{int}} \to \mathtt{t}^{\mathsf{int}}, \xi)$, where ξ is a placeholder for the set that contains the function. We need not have restricted the base language to be simply-typed: we could have chosen a rank 2 intersection type system instead. For instance, given the term $M = (\lambda f : \sigma.\ (\lambda x : \mathsf{int}.\ fI)(f0))I$ (from (Palsberg and O'Keefe 1995)) where I is the identity combinator and $\sigma = (\mathsf{int} \to \mathsf{int}) \to (\mathsf{int} \to \mathsf{int}) \wedge (\mathsf{int} \to \mathsf{int})$, we can show that M has the rank 2 intersection type $\sigma \to (\mathsf{int} \to \mathsf{int})$. Let us annotate the term in ℓPCF (writing out the identity combinator), as follows: $(\lambda^1 f.\ (\lambda^2 x.\ (f(\lambda^3 u.\ u))(f0)))(\lambda^4 v.\ v)$. Then executing the inference algorithm $\mathcal{I}$ and solving the set of flow constraints, shows that the property of M is $(\mathtt{t}^{\mathsf{int}} \to \mathtt{t}^{\mathsf{int}}, \{3\})$. This means that M evaluates to λ^3. The type of the function part of M is $(\varphi \to (\mathtt{t}^{\mathsf{int}} \to \mathtt{t}^{\mathsf{int}}, \{3\}), \{1\})$, where $\varphi = ((\mathtt{t}^{\mathsf{int}} \to \mathtt{t}^{\mathsf{int}}, \{3\}) \to (\mathtt{t}^{\mathsf{int}} \to \mathtt{t}^{\mathsf{int}}, \{3\}), \{4\}) \wedge (\mathtt{t}^{\mathsf{int}} \to \mathtt{t}^{\mathsf{int}}, \{4\})$. This shows the expected polyvariance: the two uses of f expect the identity λ^4; at one application site λ^4 calls λ^3 and returns λ^3; at the other application site it expects an integer and returns an integer.

By focusing on what set of functions every program point can possibly evaluate to, our analysis automatically performs Tang and Jouvelot's call-tracking analysis (Tang 1995): suppose an expression has the property $((\mathtt{t}^{\mathsf{int}} \to \mathtt{t}^{\mathsf{int}}, \{2\}) \to (\mathtt{t}^{\mathsf{int}} \to \mathtt{t}^{\mathsf{int}}, \{4\}), \{1\})$, then, the property signifies that it evaluates to the function labelled 1. This function when applied, may call the function labelled 2 and may yield the function labelled 4 as result. Moreover, functions labelled 2 and 4 never call any functions and are applied to integer values.

Several issues of fundamental nature have not been treated in this article and merit further investigations:

— The expressiveness of ranked intersection types for control-flow analysis needs a precise characterisation, especialy with respect to properties in the style of parametric polymorphism. The works by Mossin on Exact Flow Analysis mentioned in Section 7.2 and by Kfoury and Wells on finite-rank intersection type inference mentioned in the Introduction seem to be good starting points for such an investigation.

— The complexity (both theoretical and practical) of the analysis should be further

investigated, notably with respect to the size of the constraint sets produced by the analysis. Because of the rewriting relation $\overset{+}{\rightsquigarrow}$ (Section 6) on function properties, it is possible that the number of constraints generated due to the corresponding step in the application, conditional and recursive function cases in algorithm $\mathcal{I}$ will be exponential in the rank of function properties. However, we have not investigated optimizations, *e.g.*, on-line cycle elimination (Fähndrich et al. 1998), to cut down the size of the constraint set generated. This is a topic of future research.

— While there has been much work on control-flow analysis for functional programs in the past decade, not much attention has been paid to the use of control-flow analysis for transformations of functional programs and, more importantly, for proofs of correctness of the transformations. The main work in the area is due to Wand and his coauthors (Steckler and Wand 1997; Wand and Siveroni 1999) who have formalised the use of control-flow analysis for implementing program transformations like lightweight closure conversion, useless variable elimination, destructive array updates. Cejtin *et al.* have also used flow-directed closure conversion (Tolmach and Oliva 1998; Dimock et al. 2001) the Mlton compiler (Cejtin et al. 2000). In recent work, Banerjee *et al.* have considered a uniform method for proving the correctness of control-flow analysis-based program transformations in functional languages (Banerjee et al. 2001). The method relies upon "defunctionalisation," a mapping from a higher-order language to a first-order language. They give methods for proving defunctionalisation correct. Using this proof and common semantic techniques, they show how two program transformations—flow-based inlining and lightweight defunctionalisation—can be proven correct. However, all the techniques developed above have been for whole-program transformations. It remains a challenge to provide and formalise techniques for program transformations for program fragments and for program transformations at link time.

Acknowledgements: Thanks to the members of the Church Project, especially, Torben Amtoft, Allyn Dimock, Bob Muller, Franklyn Turbak and Joe Wells for discussions. Thanks to the anonymous referees for their suggestions and to Jens Palsberg for his interest. A special thanks to Franklyn Turbak for copious comments on the paper in a short time.

References

Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman (1986). *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1986.

Roberto Amadio and Luca Cardelli (1993). Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993.

Torben Amtoft, Flemming Nielson, and Hanne Riis Nielson (1997). Type and behaviour reconstruction for higher-order concurrent programs. *Journal of Functional Programming*, 7(3):321–347, May 1997.

Torben Amtoft, Flemming Nielson, and Hanne Riis Nielson (1999). *Type and Effect Systems: Behaviours for Concurrency.* Imperial College Press, 1999.

Torben Amtoft and Franklyn Turbak (2000). Faithful translations between polyvariant flows and polymorphic types. In G. Smolka, editor, *Proc. of European Symp. on Programming (ESOP 2000)*, Lecture Notes in Computer Science vol. 1782. Springer, 2000.

Anindya Banerjee (1997). A modular, polyvariant and type-based closure analysis. In *Proceedings of International Conference on Functional Programming (ICFP'97)*, pages 1–10, Amsterdam, The Netherlands, 1997. ACM Press.

Anindya Banerjee, Nevin Heintze, and Jon G. Riecke (2001). Design and correctness of program transformations based on control-flow analysis. In *Proceedings of International Symposium on Theoretical Aspects of Computer Software (TACS'01)*, number 2215 in Lecture Notes in Computer Science, Springer-Verlag, 2001.

Henk Barendregt (1984). *The Lambda Calculus: its Syntax and Semantics*. North-Holland, 1984.

Nick Benton (1992). Strictness logic and polymorphic invariance. In *Proceedings of the Second International Symposium on Logical Foundations of Computer Science*, number 620 in Lecture Notes in Computer Science. Springer-Verlag, 1992.

Geoff Burn, Chris Hankin, and Samson Abramsky (1986). Strictness analysis for higher-order functions. *Science of Computer Programming*, 7:249–278, 1986.

Henry Cejtin, Suresh Jagannathan, and Stephen Weeks (2000). Flow-directed closure conversion for typed languages. In G. Smolka, editor, *Proc. of European Symp. on Programming (ESOP 2000)*, Lecture Notes in Computer Science vol. 1782. Springer, 2000.

Mario Coppo, Mariangiola Dezani–Ciancaglini, and Betti Venneri (1980a). Functional characters of solvable terms. *Zeitschrifft f. Mathematische Logik*, 27:45–58, 1980.

Mario Coppo, Mariangiola Dezani–Ciancaglini, and Betti Venneri (1980b). 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.

Mario Coppo and Paula Giannini (1992). A complete type inference algorithm for simple intersection types. In Jean-Claude Raoult, editor, *Proceedings of 17th Colloquium on Trees in Algebra and Programming (CAAP '92)*, number 581 in Lecture Notes in Computer Science, pages 102–123. Springer-Verlag, 1992.

Mario Coppo and Ferruccio Damiani and Paula Giannini (1998). Inference based analysis of functional programs: dead-code and strictness. In *MSI-Memoir Volume 2, "Theories of Types and Proofs"*, pages 143–176, Mathematical Society of Japan, 1998.

Patrick Cousot and Radhia Cousot (1977). Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fix-points. In *Proceedings of the Fourth Annual ACM Symposium on Principles of Programming Languages (POPL '77)*, January 1977.

Luis Manuel Martins Damas (1985). *Type assignment in programming languages*. PhD thesis, University of Edinburgh, Edinburgh, Scotland, April 1985.

Ferruccio Damiani (1996). Refinement types for program analysis. In R. Cousot and D. Schmidt, editors, *Proceedings of 3rd International Static Analysis Symposium (SAS'96)*, number 1145 in Lecture Notes in Computer Science, pages 285–300. Springer-Verlag, 1996.

Ferruccio Damiani and Frédéric Prost (1996). Detecting and removing dead code using rank 2 intersection types. In *Proceedings of TYPES'96 Selected Papers*, number 1512 in Lecture Notes in Computer Science, pages 66–87. Springer-Verlag, 1998.

Allyn Dimock, Robert Muller, Franklyn Turbak, and Joseph B. Wells (1997). Strongly typed flow-directed representation transformations. In *Proceedings of International Conference on Functional Programming (ICFP'97)*, pages 11–24, Amsterdam, The Netherlands, 1997. ACM Press.

Allyn Dimock, Ian Westmacott, Robert Muller, Franklyn Turbak, Joe Wells, and Jeffrey Considine (2000). Program representation size in an intermediate language with intersection and union types. In *Proceedings of the third workshop on Types in Compilation, (TIC '00)*, number 2071 in Lecture Notes in Computer Science, editor Robert Harper. Springer-Verlag, 2001.

Allyn Dimock, Ian Westmacott, Robert Muller, Franklyn Turbak, and Joe Wells (2001). Functioning without closure: type-safe customized function representations for Standard ML. In *Proceedings of the International Conference on Functional Programming (ICFP '01)*. ACM Press.

Dirk Dussart, Fritz Henglein, and Christian Mossin (1995). Polymorphic recursion and subtype qualifications: Polymorphic binding-time analysis in polynomial time. In A. Mycroft, editor, *Proceedings of the Static Analysis Symposium (SAS '95)*, number 983 in Lecture Notes in Computer Science. Springer-Verlag, 1995.

Manuel Fähndrich and Jeffrey S. Foster and Zhendong Su and Alexander Aiken (1998). Partial online cycle elimination in inclusion constraint graphs. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI '98)*, pages 85–96. ACM Press, 1998.

Chris Hankin and Daniel Le Métayer (1994). Deriving algorithms from type inference systems: Applications to strictness analysis. In *Proceedings of Twentyfirst ACM Symposium on Principles of Programming Languages (POPL'94)*, pages 202–212. ACM Press, 1994.

Chris Hankin and Daniel Le Métayer (1995). Lazy type inference and program analysis. *Science of Computer Programming*, 25:219–249, 1995.

John Hannan and Dale Miller (1992). From operational semantics to abstract machines. *Mathematical Structures in Computer Science*, 2(4):415–459, 1992.

John Hatcliff and Olivier Danvy (1997). A computational formalization for partial evaluation. *Mathematical Structures in Computer Science*, 7:507–541, 1997. Special issue containing selected papers presented at the 1995 Workshop on Logic, Domains, and Programming Languages, Darmstadt, Germany.

Nevin Heintze (1995). Control-flow analysis and type systems. In Alan Mycroft, editor, *Proceedings of Static Analysis Symposium*, number 983 in Lecture Notes in Computer Science, pages 189–206. Springer-Verlag, 1995.

Fritz Henglein and Christian Mossin (1994). Polymorphic binding-time analysis. In *Proceedings of the Fifth European Symposium on Programming (ESOP '94)*, Lecture Notes in Computer Science. Springer-Verlag, 1994.

Suresh Jagannathan and Stephen Weeks (1995). A unified treatment of flow analysis in higher-order languages. In *Proceedings of the Twentysecond Annual ACM Symposium on Principles of Programming Languages (POPL '95)*, San Francisco, California, January 1995.

Suresh Jagannathan, Stephen Weeks, and Andrew Wright (1997). Type-directed flow analysis for typed intermediate languages. In P. v. Hentenryck, editor, *Proceedings of the Static Analysis Symposium (SAS'97)*, number 1302 in Lecture Notes in Computer Science, pages 232–249. Springer-Verlag, 1997.

Thomas Jensen. Strictness analysis in logical form (1991). In J. Hughes, editor, *Proceedings of 5th ACM Conference on Functional Programming Languages and Computer Architecture*, number 523 in Lecture Notes in Computer Science, pages 352–366. Springer Verlag, 1991.

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

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

Thomas Jensen (1998). Inference of polymorphic and conditional strictness properties. In *Proc. of 25th ACM Symposium on Principles of Programming Languages*, pages 209–221. ACM Press, 1998.

Trevor Jim (1995). Rank 2 type systems and recursive definitions. Technical Memorandum MIT/LCS/TM-531, Laboratory for Computer Science, Massachussetts Institute of Technology, November 1995.

Trevor Jim (1996). What are principal typings and what are they good for? In *Proceedings of the Twentythird Annual ACM Symposium on Principles of Programming Languages (POPL '96)*, St. Petersburg, Florida, January 1996.

Neil D. Jones (1981). Flow analysis of lambda expressions. In *Proceedings of Eighth Colloquium on Automata, Languages, and Programming*, number 115 in Lecture Notes in Computer Science. Springer-Verlag, 1981.

Neil D. Jones (1987). Flow analysis of lazy, higher order functional programs. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*. Ellis Horwood, 1987.

Assaf J. Kfoury and Joseph B. Wells (1999). Principality and decidable type inference for finite-rank intersection types. In *Proceedings of the Twentysixth Annual ACM Symposium on Principles of Programming Languages (POPL '99)*, San Antonio, Texas, January 1999.

Naoki Kobayashi (2000). Type-based useless variable elimination. In Julia L. Lawall, editor, *Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 84–93, Boston, Massachusetts, January 2000. ACM, ACM Press.

Tsung-Min Kuo and Prateek Mishra (1989). Strictness analysis : A new perspective based on type inference. In *Proceedings of 4th. International Conference on Functional Programming and Computer Architecture*. ACM Press, 1989.

Daniel Leivant (1983). Polymorphic type inference. In *Proceedings of the Tenth Annual ACM Symposium on Principles of Programming Languages (POPL '83)*, pages 88–98, January 1983.

John C. Mitchell (1991). Type inference with simple subtypes. *Journal of Functional Programming*, 1(3):245–285, 1991.

Christian Mossin (1997a). Exact flow analysis. In *Proceedings of the Fourth International Static Analysis Symposium*, Paris, France, September 1997.

Christian Mossin (1997b). *Flow analysis of typed higher-order programs*. PhD thesis, DIKU, University of Copenhagen, January 1997.

Flemming Nielson and Hanne Riis Nielson (1992). *Two-Level Functional Languages*, volume 34 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1992.

Flemming Nielson and Hanne Riis Nielson (1997). Infinitary control flow analysis: a collecting semantics for closure analysis. In *Proceedings of the Twentyfourth Annual ACM Symposium on Principles of Programming Languages (POPL '97)*, Paris, France, January 1997.

Hanne Riis Nielson and Flemming Nielson (1998). Automatic binding time analysis for a typed λ-calculus. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages (POPL '88)*, pages 98–106, January 1988.

Jens Palsberg (1995). Closure analysis in constraint form. *ACM Transactions on Programming Languages and Systems*, 17(1):47–62, January 1995.

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

Jens Palsberg and Christina Pavlopoulou (1998). From polyvariant flow information to intersection and union types. *Journal of Functional Programming*, 11(3):263–317, May 2001.

Jens Palsberg and Michael Schwartzbach (1995). Safety analysis versus type inference. *Information and Computation*, 118(1):128–141, 1995.

Gordon D. Plotkin (1977). LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

The Church project. `http://types.bu.edu/comp-flow-types.html`.

Jon G. Riecke (1991). *The Logic and Expressibility of Simply-Typed Call-by-Value and Lazy Languages*. PhD thesis, Massachusetts Institute of Technology, 1991. Available as technical report MIT/LCS/TR-523 (MIT Laboratory for Computer Science).

Patrick Sallé (1978). Une extension de la théorie des types en λ-calcul. In G. Ausiello and C. Böhm, editors, *Fifth International Conference on Automata, Languages and Programming*, pages 398–410. Springer-Verlag, 1978.

David A. Schmidt (1995). Natural semantics-based abstract interpretation. In Alan Mycroft, editor, *Proceedings of the Static Analysis Symposium*, number 983 in Lecture Notes in Computer Science, pages 1–18. Springer-Verlag, 1995.

Peter Sestoft (1988). Replacing function parameters by global variables. Master's thesis, University of Copenhagen, 1988.

Peter Sestoft (1991). *Analysis and Efficient Implementation of Functional Programs*. PhD thesis, DIKU, Copenhagen, Denmark, October 1991. Rapport Nr. 92/6.

Olin Shivers (1991). *Control-Flow Analysis of Higher-Order Languages or Taming Lambda*. PhD thesis, Carnegie-Mellon University, Pittsburgh, Pennsylvania, May 1991. Technical Report CMU-CS-91-145.

Paul A. Steckler and Mitchell Wand (1997). Lightweight closure conversion. *ACM Transactions on Programming Languages and Systems*, 19(1):48–86, 1997.

Jean-Pierre Talpin and Pierre Jouvelot (1994). The type and effect discipline. *Information and Computation*, 2(111):245–296, 1994.

Yan-Mei Tang (1994). *Systèmes d'effet et interprétation abstraite pour l'analyse de flot de contrôle*. PhD thesis, Ecole Nationale Supérieure des Mines de Paris, March 1994. Rapport A/258/CRI.

Yan-Mei Tang and Pierre Jouvelot (1995). Effect systems with subtyping. In William L. Scherlis, editor, *Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, La Jolla, California, June 1995. ACM SIGPLAN, ACM Press.

Mads Tofte and Jean-Pierre Talpin (1994). Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Proc. of 21st Symposium on Principles of Programming Languages (POPL'94)*. ACM Press, 1994.

Mads Tofte and Jean-Pierre Talpin (1997). Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

Andrew Tolmach and Dino Oliva (1998). From ML to Ada: strongly-typed language interoperability via source translation. *Journal of Functional Programming*, 8(4):367–412, 1998.

Steffen van Bakel (1992). Complete restrictions of the intersection type discipline. *Theoretical Computer Science*, 102:135–163, 1992.

Steffen van Bakel (1993). Intersection type disciplines in lambda calculus and applicative term rewriting systems. PhD thesis, Mathematisch Centrum, Amsterdam, 1993.

Steffen van Bakel (1996). Rank 2 intersection type assignment in term rewriting systems. *Fundamenta Informaticae*, 26(2), 1996.

Mitchell Wand and Igor Siveroni (1999). Constraint systems for useless variable elimination. In *Proceedings of the Twentysixth Annual ACM Symposium on Principles of Programming Languages*, pages 291–302, January 1999.

Andrew Wright and Matthias Felleisen (1991). A syntactic approach to type soundness. *Information and Computation*, 3(2):181–210, 1991.