A Modular, Polyvariant, and Type-Based Closure Analysis *

Anindya Banerjee
Xinotech Research†
ab@xinotech.com

Abstract

We observe that the *principal typing property* of a type system is the enabling technology for *modularity* and *separate compilation* [10]. We use this technology to formulate a modular and polyvariant closure analysis, based on the rank 2 intersection types annotated with control-flow information.

Modularity manifests itself in a syntax-directed, annotated-type inference algorithm that can analyse *program fragments* containing free variables: a *principal typing* property is used to formalise it. Polyvariance manifests itself in the separation of different behaviours of the same function at its different uses: this is formalised via the rank 2 intersection types. As the rank 2 intersection type discipline types at least all (core) ML programs, our analysis can be used in the separate compilation of such programs.

1 Introduction

Compiler optimisations are crucially dependent on the availability of control-flow information at compile time. For any first-order imperative program, this 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 [1]. For higher-order programs, however, a control-flow graph is often unavailable. The reason is that functions in a higher-order program are treated as data — they can be passed between two program points via function calls and can arise as the result of evaluation at any program point. Thus one needs to *calculate* flow information — it cannot be deciphered from an inspection of the structure of the program text. Hence, a number of *control-flow analyses* have been proposed [8, 11, 20] for higher-order programs. Since functions are traditionally implemented as closures, such analyses are also called *closure analyses* [19].[1] All of them answer the following interrelated questions: what are the possible functions called at a program point? and: what are the possible functions that can be the result of evaluation at a program point?

1.1 Techniques for closure analysis

The classical technique for closure analysis is *abstract interpretation* [4] of either the denotational semantics of the language [20] or of its operational semantics [14]. An equivalent technique uses a system of constraints to specify these analyses so that flow information is obtained as the minimal solution of the constraint system [16]. Both techniques are *global* in that they require the availability of the *entire program* before analysis. Further, they assume the program to be *closed*, or, if open, to have simple (*e.g.*, integer or boolean), non-function inputs.

Recently, there has been a surge of interest in using annotated type systems for program analysis. The intuition is that we can annotate types and expressions in a language with the properties of interest, *e.g.*, flows [23], binding times, strictness and totality [21], effects [22], regions [24], concurrent behaviours [15] *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. [2]

For closure analysis, we annotate every function in an expression with a distinct label, and associate a set of function labels with every function type. The intuition is that if an expression, e, has the function type, τ, with *flow annotation*, ϕ, then its *evaluation* will yield a closure, whose text is a function having a label predicted by ϕ. The *static* determination of the set of function labels that e can possibly evaluate to, is thus synonymous with the calculation, via an *annotated-type inference algorithm*, of its flow annotations.

While the type-based approach maynot be as precise as the abstract interpretation-based approach, it nonetheless outlines a method for performing *local* and more importantly, *modular* program analysis. "Local" means that the analysis of an expression is derived

*To appear in ICFP 97.

†Minneapolis, MN 55414, USA. Phone: (+1)(612)379-3844, Fax: (+1)(612)379-3833. This work was supported by a postdoctoral fellowship at the Laboratoire d'Informatique, Ecole Polytechnique, Palaiseau, France, (thanks to Radhia Cousot) and subsequently at the Computer Science Department, Aarhus University, Denmark, under the DART project of the Danish National Research Councils (thanks to Flemming Nielson).

[1]In the rest of the paper we will use these terms interchangeably.

[2]There is of course the issue of decidability of type inference in the underlying type system: for instance, it is well known that type inference is undecidable in System F or the intersection type discipline.

through the composition of the analyses of its proper sub-parts. We say that an analysis is "modular" if it can analyse *program fragments containing free variables*, *i.e.*, *modules*, in isolation, and if the linking of fragments does not require their re-analysis. A modular program analysis is thus indispensable in the context of *separate compilation*.

1.2 Related work on type systems and closure analysis

The inspiration for this paper comes from three sources:

- Palsberg and O'Keefe show that Amadio and Cardelli's type system with recursive types and subtyping [2] is equivalent to *safety analysis* [17, 18].[3] 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. Suppose we know that an expression is typable; is there an algorithm to uncover its type-derivation tree? The answer is "yes", and the method involves performing a safety analysis of the expression.

- Heintze shows that given a variety of type systems instrumented by control-flow information — the instrumented types are termed *control-flow types* — there exist corresponding closure analyses augmented by type information such that each type system is equivalent to its corresponding closure analysis [7]. Equivalence here means that each system calculates the same flow and type information.

Both of the above papers address type systems for closure analysis, but their methods of computation of flow information are as follows: 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 we obtain *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 λ-calculus with a recursion operator [23]. 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, TE, 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. In fact, this is the general technique that is used for all the example program analyses mentioned in Section 1.1. The technique, however, is *non-modular*: it requires the user to specify the annotated types of the free variables of e as is manifest in the parameters to the inference algorithm.

[3] Safety analysis for the untyped λ-calculus is basically a constraint-based, global program analysis that is used to prevent run-time errors due to misuse of constants (as in *e.g.*, the application, $0(\lambda x.x)$) —*without type reconstruction*.

1.3 This paper

A curious observation due to Damas [5, Chapter 1], is that the simply typed λ-calculus satisfies the *principal typing property*: given a typable program fragment, e, possibly containing free variables, there is a pair, $\langle TE,\ \tau\rangle$, such that $TE \vdash e : \tau$ 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; secondly, when fragments are linked, the typing of e might possibly change — but principality guarantees that the new typing is always an instance of $TE \vdash e : \tau$. Thus we can avoid a re-inference of e upon linking. Principal typing is therefore crucial to ensuring modularity. In the rest of the paper, we apply this property to obtain a modular closure analysis.

Our goal is first, to provide a *modular closure analysis* of program fragments in languages with type systems at least as powerful as (core) ML's. Secondly, we are also interested in making our analysis *polyvariant*, *i.e.*, a function can have different *behaviours* (*i.e.*, be applied to different functions) at its different uses.

Note that in an ML-like language with polymorphic type schemes, *the principal typing property no longer holds* [5, Chapter 2]. Instead, we obtain the weaker notion of principal type schemes: if a type scheme can be inferred for an expression *from a particular set of assumptions*,[4] then the expression has a principal type scheme *under those assumptions*. This entails that we cannot infer a principal type scheme for a *program fragment with free variables*, unless supplied the type schemes of the free variables; this, in turn, entails that we ought to know the *definitions* of the free variables so that we can type the *uses* of these definitions. This runs contrary to our expectations that we must be able to type all uses of (free) variables in a module independently of their definitions, which may reside in a different module. Program analyses based on the ML type system are therefore often non-modular.

The following question [17, pp.579, 591] is open: "Does there exist a direct, (*i.e.*, modular) way of computing flow information in Amadio and Cardelli's system without a reduction to safety analysis"? In this paper we answer this question positively for another very expressive type system, namely, the rank 2 intersection type system. We thus address both of our goals: indeed, Jim has shown that this system can type more terms than ML while retaining the same complexity of type inference [10]. He has further demonstrated the usefulness of this type system in typing recursive definitions, in accurate error detection due to type mismatches, in incremental compilation, *etc.*

[4] a proviso *not* required for the simply-typed λ-calculus.

We develop a modular, polyvariant and type-based closure analysis based on the rank 2 intersection type system with subtyping. Annotating types with flows, we show that there exists a sound and complete type inference algorithm that can compute control-flow information directly from the structure of the program text. Subtyping is used to orient flow information; the *principal typing* property of the annotated type system lends *modularity* to the analysis, so that program fragments with free variables can be separately analysed; and the polymorphism inherent in the type system renders the analysis *polyvariant*, so that the same function can be specified to have different behaviours at its different uses.

Polyvariance in our framework is similar to the notion of *polymorphic splitting* advocated by Jagannathan and Wright for improving the precision of 0-CFA [9]. Our work differs in that polyvariance is obtained as a result of the inherent polymorphism in the rank 2 intersection type system rather than as a result of an explicit polyvariant closure analysis based on abstract interpretation.

1.4 Organisation of the paper

Section 2 briefly reviews closure analysis and gives a couple of examples. Section 3 introduces the system $\mathbf{L_2}$, which is the proposed type system for closure analysis. Section 4 and its subsections introduce the framework for type inference in $\mathbf{L_2}$, the crucial notion of matching, and the inference algorithm. Properties of the inference algorithm are stated in Section 5. Section 6 shows an example where the inference algorithm is applied. A discussion of the framework follows in Section 7. Section 8 concludes.

2 Closure analysis

Consider an expression in the (call-by-value) λ-calculus. We will label all λ-abstractions and variables in this expression uniquely, so that $\lambda^\ell x$ binds occurrences of x^ℓ. We often refer to a λ-abstraction by its label.

A node in the syntax tree of an expression is called a "program point". Then, given a closed expression, closure analysis seeks to answer the following questions:

- What set of labelled lambdas can each program point *evaluate* to?
- What set of labelled lambdas can each λ-abstraction in the expression be *applied* to?

Suppose the analysis detects that one of the λ-abstractions that ℓ can be applied to is ℓ'. Then this is analogous to saying that the bound variable, x^ℓ, say, of ℓ can be bound to ℓ'. The second question above can thus be reformulated as:

- What set of labelled lambdas can every variable in the closed expression be *bound* to?

Let us give a couple of examples of closure analysis. For exact details of the usual abstract interpretation-based analysis, we refer the reader to the work of Sestoft [19] and Shivers [20].[5]

[5] Mossin has recently discovered that Sestoft's analysis is *not* Shivers' 0-CFA, as is commonly mentioned in the literature. For examples, see his forthcoming Phd thesis [13, chapter 8]. The analysis presented in this paper, is, in Mossin's parlance, a "value-flow analysis", where we are interested only in closure values.

2.1 Examples

We wish to analyse the following terms:

(I) $(\lambda^1 g\,.\,g(g(\lambda^2 v\,.\,v)))(\lambda^3 x\,.\,\lambda^4 y\,.\,y)$.
The analysis 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 $\emptyset$.
   $\lambda^4 y\,.\,y$, yields $\{4\}$.
3. The entire application expression yields $\{4\}$.

The interesting case is that of x: it yields both $\{2\}$ and $\{4\}$! This is because, $\lambda^3 x\,.\,\lambda^4 y\,.\,y$ once gets applied to $\lambda^2 v\,.\,v$ and again to the result of this application, *i.e.*, to $\lambda^4 y\,.\,y$.

(II) $(\lambda^1 z\,.\,zz)(\lambda^2 y\,.\,y)$. The analysis yields the following results:

1. The function part of the application, $\lambda^1 z\,.\,zz$, yields $\{1\}$.
   z yields $\{2\}$.
2. The argument part of the application, $\lambda^2 y\,.\,y$, yields $\{2\}$.
   y yields $\{2\}$.
3. The entire application expression yields $\{2\}$.

One can verify the above results by reducing the expressions. Note that the first example is typable in the simply-typed λ-calculus and in ML, whereas the second is not. Thus a type-based analysis based on the simple-type system, where properties are represented using simple-type constructors, cannot be used to do closure analysis for the second example. On the contrary, an abstract interpretation-based analysis would succeed.

For the first example, note that a monovariant analysis would analyse x and report that it possibly evaluates to the set, $\{2, 4\}$. For a polyvariant analysis, however, we should rather have a formalism that says: at its innermost occurrence, x is bound to the λ-abstraction labelled 2, and at its outer occurrence, x is bound to the λ-abstraction labelled 4. Informally, we could say that g is bound to the λ-abstraction labelled 3, which has type "$\{2\} \to \{4\}$" at the former occurrence, and has type "$\{4\} \to \{4\}$" at the latter occurrence. We cannot "unify" these types, instead should maintain their identities by resorting to an intersection type — "$\{2\} \to \{4\} \wedge \{4\} \to \{4\}$".

3 A type system for polyvariant closure 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 [12] and extended to behaviour analysis by Amtoft, Nielson and Nielson [3, 15]. We show that the rank 2 fragment of the intersection type discipline can be instrumented to perform a polyvariant closure analysis. We will call this instrumented system $\mathbf{L_2}$.

A type, τ, is of rank k, if no intersection type constructor, $\wedge$, occurs to the left of k arrows in τ. We restrict ourselves to rank 2, because typability for ranks higher than 2 is an open question, and in the full intersection type discipline typability is undecidable; further, the rank 2 system can type more terms than ML and has the principal typing property [10].

The *terms* of the system are the terms of the λ-calculus. The *types* of the system are classified as follows:

$$\begin{aligned}\mathbf{T_0} = &\{t \mid t \text{ is a type variable}\} \cup \\ &\{\sigma \xrightarrow{\phi} \tau \mid \sigma, \tau \in \mathbf{T_0}, \phi \in \mathsf{Flow}\}^6\end{aligned}$$

$$\mathbf{T_1} = \mathbf{T_0} \cup \{\sigma \wedge \tau \mid \sigma, \tau \in \mathbf{T_1}\}$$

$$\mathbf{T_2} = \mathbf{T_0} \cup \{\sigma \xrightarrow{\phi} \tau \mid \sigma \in \mathbf{T_1},\ \tau \in \mathbf{T_2}, \phi \in \mathsf{Flow}\}$$

$\mathbf{T_0}$ is the set of *simple* or *rank 0 types*. The set $\mathbf{T_1}$ is the set containing *rank 1 intersection types*: *i.e.*, it contains $\mathbf{T_0}$ as well as finite, non-empty intersections of simple types. A $\mathbf{T_1}$ type can be written in the notation $\bigwedge_{i\in I}(\tau_i)$, where each τ_i is in $\mathbf{T_0}$. The set $\mathbf{T_2}$ contains $\mathbf{T_0}$ and the set of function types which are a *subset* of the *rank 2 intersection types*, and in which the intersection type constructor is restricted to appear only to the left of atmost one arrow. Clearly, $\mathbf{T_1} \not\subseteq \mathbf{T_2}$ and $\mathbf{T_0} = \mathbf{T_1} \cap \mathbf{T_2}$.

The set Flow denotes flows: a flow, ϕ, has the BNF, $\phi ::= \kappa \mid \xi \mid \phi_1 \cup \phi_2$, where $\kappa \in \mathcal{P}(Labels)$ and ξ is a *flow variable*.

3.1 Subtyping

Corresponding to the sets $\mathbf{T_i}$ above, we have the subtyping relations, $\leq_i$, for $i \in \{0, 1, 2\}$. A set, K, of type constraints is said to be *atomic*, iff all type constraints in K are of the form, $t \leq_0 t'$, where, t, t' are type variables. A set, C, of flow constraints is said to be *regular*, iff all flow constraints in C are of the form, $\xi \succeq \phi$, where ξ is a flow variable.

Subtyping judgements take the form: $K, C \vdash_{\mathsf{ST}} \sigma \leq_i \tau$, where K is a set of *atomic type constraints*, C is a set of *regular flow constraints*, and $i \in \{0, 1, 2\}$. The judgement should be read as: under the atomic type constraints, K, and the regular flow constraints, C, we can deduce that the rank i-type σ is a subtype of the rank i-type τ, for $i \in \{0, 1, 2\}$. Since function types are annotated with flows, subtyping on types requires a comparison of flows too. Thus the judgement $K, C \vdash_{\mathsf{FL}} \phi' \succeq \phi$ means that under the atomic type constraints, K, and the regular flow constraints, C, the flow, ϕ', is less precise than the flow, ϕ. Figure 1 provides a formal definition of subtyping and flow comparisons.[7]

$$K, C \vdash_{\mathsf{FL}} \kappa' \succeq \kappa, \quad \mathit{if}\ \kappa' \supseteq \kappa$$

$$K, C \vdash_{\mathsf{FL}} \xi \succeq \phi, \quad \mathit{if}\ (\xi \succeq \phi) \in C$$

$$\frac{\begin{array}{c} K, C \vdash_{\mathsf{FL}} \phi \succeq \phi' \\ K, C \vdash_{\mathsf{FL}} \phi' \succeq \phi'' \end{array}}{K, C \vdash_{\mathsf{FL}} \phi \succeq \phi''}$$

$$K, C \vdash_{\mathsf{ST}} \sigma \leq_i \sigma, \quad i \in \{0, 1, 2\}$$

$$\frac{\begin{array}{c} K, C \vdash_{\mathsf{ST}} \sigma \leq_i \tau \\ K, C \vdash_{\mathsf{ST}} \tau \leq_i \rho \end{array}}{K, C \vdash_{\mathsf{ST}} \sigma \leq_i \rho} \quad i \in \{0, 1, 2\}$$

$$K, C \vdash_{\mathsf{ST}} t \leq_0 t', \quad \text{if } (\mathrm{t} \leq_0 \mathrm{t}') \in K$$

$$\frac{\begin{array}{c} K, C \vdash_{\mathsf{ST}} \sigma_1 \leq_0 \tau_1 \\ K, C \vdash_{\mathsf{ST}} \tau_2 \leq_0 \sigma_2 \\ K, C \vdash_{\mathsf{FL}} \phi' \succeq \phi \end{array}}{K, C \vdash_{\mathsf{ST}} (\tau_1 \xrightarrow{\phi} \tau_2) \leq_0 (\sigma_1 \xrightarrow{\phi'} \sigma_2)}$$

$$\frac{\begin{array}{l} \forall \tau \in \{\tau_j \mid j \in J\}.\ \exists \sigma \in \{\sigma_i \mid i \in I\}. \\ K, C \vdash_{\mathsf{ST}} \sigma \leq_0 \tau \end{array}}{K, C \vdash_{\mathsf{ST}} \bigwedge_{i\in I}(\sigma_i) \leq_1 \bigwedge_{j\in J}(\tau_j)}$$

$$\frac{K, C \vdash_{\mathsf{ST}} \sigma \leq_0 \tau}{K, C \vdash_{\mathsf{ST}} \sigma \leq_2 \tau}$$

$$\frac{\begin{array}{c} K, C \vdash_{\mathsf{ST}} \sigma_1 \leq_1 \tau_1 \\ K, C \vdash_{\mathsf{ST}} \tau_2 \leq_2 \sigma_2 \\ K, C \vdash_{\mathsf{FL}} \phi' \succeq \phi \end{array}}{K, C \vdash_{\mathsf{ST}} (\tau_1 \xrightarrow{\phi} \tau_2) \leq_2 (\sigma_1 \xrightarrow{\phi'} \sigma_2)}$$

Figure 1: Subtyping in $\mathbf{L_2}$

The typing rules of $\mathbf{L_2}$ follow in Figure 2. For $i \in \{0, 1, 2\}$, we say that a type environment TE is a $\mathbf{T_i}$ type environment, iff for all $x \in \mathbf{dom}(TE)$, $TE(x)$ is a $\mathbf{T_i}$ type. All type environments in $\mathbf{L_2}$ are $\mathbf{T_1}$ type environments whereas derived types are in $\mathbf{T_2}$. The subtyping relation, $\leq_1$ is extended to $\mathbf{T_1}$ type environments so that $TE \leq_1 TE'$ iff for all $x \in \mathbf{dom}(TE')$, it

[6] Base types like *int*, *real*, *bool*, *etc.*, belong to $\mathbf{T_0}$. For simplicity, we omit them in this paper.

[7] In the sequel, we will often need judgements of the form, $K, C \vdash_{\mathsf{ST}} K'$ and $K, C \vdash_{\mathsf{FL}} C'$, where, K' and C' are sets of type constraints and flow constraints. They have the obvious meanings.

is the case that $x \in \mathbf{dom}(TE)$ and $TE(x) \leq_1 TE'(x)$. A typing is of the form, $K, C, TE \vdash e : \tau$, and is read: under the atomic type constraints, K, the regular flow constraints, C, and the $\mathbf{T_1}$-type environment, TE, the expression, e, has the $\mathbf{T_2}$-type, τ.

$$K, C, TE \oplus (x^\ell : \bigwedge_{i \in I} \tau_i) \vdash x^\ell : \tau_{i_0}, \text{ if } i_0 \in I$$

$$\frac{K, C, TE \oplus (x^\ell : \tau_1) \vdash e : \tau_2}{K, C, TE \vdash \lambda^\ell x . e : (\tau_1 \xrightarrow{\kappa} \tau_2)} \quad \ell \in \kappa$$

$$\frac{\begin{array}{l} K, C, TE \vdash e_1 : (\bigwedge_{i \in I} \tau_i) \xrightarrow{\phi} \tau \\ (\forall i \in I).\ K, C, TE \vdash e_2 : \tau_i \end{array}}{K, C, TE \vdash e_1 e_2 : \tau}$$

$$\frac{\begin{array}{l} K, C, TE \vdash e : \tau \\ K, C \vdash_{\mathsf{ST}} \tau \leq_2 \sigma \end{array}}{K, C, TE \vdash e : \sigma}$$

Figure 2: The system $\mathbf{L_2}$

4 Type inference in $\mathbf{L_2}$

4.1 Preliminaries

Given an expression, e, we wish to infer a set of atomic type constraints, K, a set of regular flow constraints, C, and a $\mathbf{T_1}$ type environment, TE, under which e can be assigned a $\mathbf{T_2}$ type, τ. We are especially interested in τ of the form $\tau_1 \xrightarrow{\phi} \tau_2$, where ϕ estimates the flow information, *i.e.*, the set of all closures that e can possibly evaluate to.

In preparation for the type inference algorithm, we stipulate that *substitutions map type variables to $\mathbf{T_0}$ types, and flow variables to flows* — the restriction to $\mathbf{T_0}$ types assuring that all inferred types are $\mathbf{T_2}$ types. We will also need subtypings between $\mathbf{T_2}$ types and $\mathbf{T_1}$ types. The reason is that if we have an application of the form $e_0 e_1$, and the type of e_0 is inferred to be the $\mathbf{T_2}$ type, $\tau \xrightarrow{\phi} \tau'$, and the type of e_1 is inferred to be the $\mathbf{T_2}$ type, τ_1, then τ_1 must be a subtype of τ. But since $\tau \xrightarrow{\phi} \tau'$ is a $\mathbf{T_2}$ type, τ is a $\mathbf{T_1}$ type! Accordingly, we have the following definition of the subtyping relation $\leq_{2,1}$.

Definition 1 ($\leq_{2,1}$) *Let σ be a $\mathbf{T_2}$ type and τ_i, a $\mathbf{T_0}$ type, for all $i \in I$. Further, let K be a set of atomic type constraints, and C be a set of regular flow constraints. Then, $K, C \vdash_{\mathsf{ST}} \sigma \leq_{2,1} \bigwedge_{i \in I} \tau_i$, iff $K, C \vdash_{\mathsf{ST}} \sigma \leq_2 \tau_i$, for all $i \in I$.* [8]

Mitchell has observed that the key to generating atomic type constraints from a subtype relation on types σ and τ, say, is to make σ and τ *match* [12], *i.e.*, to force them to have the same *shape* [3]. Here we run into a technical difficulty due to the presence of intersection types. As before, let us consider an application of the form, $e_0 e_1$. Suppose the type of e_0 is inferred to be the $\mathbf{T_2}$ type, $\sigma_0 \wedge \sigma_1 \xrightarrow{\phi} \sigma_2$, and that of e_1, the $\mathbf{T_2}$ type, $\tau_0 \wedge \tau_1 \xrightarrow{\phi'} \tau_2$. If the application is to successfully type-check, then clearly, $\tau_0 \wedge \tau_1 \xrightarrow{\phi'} \tau_2 \leq_{2,1} \sigma_0 \wedge \sigma_1$. By definition of the $\leq_{2,1}$ relation, this implies that σ_0, σ_1 are $\mathbf{T_0}$ types and, in particular, that $\tau_0 \wedge \tau_1 \xrightarrow{\phi'} \tau_2 \leq_{2,1} \sigma_0$. Now if we directly match σ_0 with $\tau_0 \wedge \tau_1 \xrightarrow{\phi'} \tau_2$, then σ_0 is forced to have the shape of a $\mathbf{T_2}$ type that is *not a* $\mathbf{T_0}$ *type*, leading to a contradiction. We will see later that the solution to this difficulty is to transform any $\leq_{2,1}$ subtyping into a set of $\leq_0$ subtypings and then match on the latter set.

4.1.1 Matching

Adapting Mitchell's notion, we say that two $\mathbf{T_0}$ types, σ and τ *match* iff they have the same "shape". This is formalised in the following definition.

Definition 2 (Matching for $\mathbf{T_0}$ types) *Two $\mathbf{T_0}$ types, σ and τ match iff any of the following hold:*

- *Both σ and τ are type variables.*
- *$\sigma = \tau$.*
- *If σ has form, $\sigma_1 \xrightarrow{\phi} \sigma_2$ and τ has form, $\tau_1 \xrightarrow{\phi'} \tau_2$, then it is the case that σ_1 and τ_1 match and σ_2 and τ_2 match.*

Now let $\bigwedge_{i \in I}(\sigma_i)$ and $\bigwedge_{j \in J}(\tau_j)$ be two $\mathbf{T_1}$ types. Clearly, each of the σ_i's and each of the τ_j's are $\mathbf{T_0}$ types. We then have the following definition of matching for $\mathbf{T_1}$ types.

Definition 3 (Matching for $\mathbf{T_1}$ types) *Two $\mathbf{T_1}$ types, $\bigwedge_{i \in I}(\sigma_i)$ and $\bigwedge_{j \in J}(\tau_j)$ match iff for all $\tau \in \{\tau_j \mid j \in J\}$, there exists $\sigma \in \{\sigma_i \mid i \in I\}$, such that σ and τ match.*

Note that Definition 3 does not necessarily imply that two matching $\mathbf{T_1}$ types have the same "shape".

Finally, we have the following definition of matching for $\mathbf{T_2}$ types — note that here again, two matching $\mathbf{T_2}$ types do not necessarily imply that they have the same "shape".

Definition 4 (Matching for $\mathbf{T_2}$ types) *Two $\mathbf{T_2}$ types, σ and τ match iff either:*

- *σ and τ are both $\mathbf{T_0}$ types and they match according to Definition 2, or,*
- *σ has form, $\sigma_1 \xrightarrow{\phi} \sigma_2$ and τ has form, $\tau_1 \xrightarrow{\phi'} \tau_2$, and the $\mathbf{T_1}$ types, τ_1 and σ_1 match, and the $\mathbf{T_2}$ types, σ_2 and τ_2 match.*

The three definitions above inspire the following fact:

Fact 1 *Let K be a set of atomic type constraints, C be a set of regular flow constraints and σ and τ be two $\mathbf{T_i}$ types such that the judgement $K, C \vdash_{\mathsf{ST}} \sigma \leq_i \tau$ can be asserted for $i \in \{0, 1, 2\}$. Then σ and τ are matching.*

[8] Note that since τ_i is a $\mathbf{T_0}$ type, it is a $\mathbf{T_2}$ type and hence it is legal to write $\sigma \leq_2 \tau_i$.

Since we need to deal with the $\leq_{2,1}$ relation in the inference algorithm, we need to define what it means for a $\mathbf{T_2}$ type and a $\mathbf{T_1}$ type to match.

Definition 5 (Matching $\mathbf{T_2}$ and $\mathbf{T_1}$ types) *Let σ be a $\mathbf{T_2}$ type and $\bigwedge_{i\in I}(\tau_i)$, a $\mathbf{T_1}$ type. Then σ matches $\bigwedge_{i\in I}(\tau_i)$, provided σ matches the $\mathbf{T_0}$ (hence the $\mathbf{T_2}$) type, τ_i, for all $i \in I$.*

The following fact is immediate from Definition 5 and Fact 1:

Fact 2 *Let K be a set of atomic type constraints, C be a set of regular flow constraints, σ be a $\mathbf{T_2}$ type and $\bigwedge_{i\in I}(\tau_i)$ be a $\mathbf{T_1}$ type such that the judgement, $K, C \vdash_{\mathsf{ST}} \sigma \leq_{2,1} \bigwedge_{i\in I}(\tau_i)$ can be asserted. Then σ and $\bigwedge_{i\in I}(\tau_i)$ are matching.*

Consider the scenario where the types σ and τ are a possibly non-matching pair, such that, either, both σ and τ are $\mathbf{T_i}$ types, ($i \in \{0,1,2\}$), or, σ is a $\mathbf{T_2}$ type and τ is a $\mathbf{T_1}$ type. If there is a substitution, S, that makes $S\sigma$ and $S\tau$ match, then such a substitution is termed a *matching substitution* for σ and τ. The notion of a matching substitution is extended to a set of possibly non-matching pairs of types in the obvious manner.

4.1.2 Matching problems

Consider a possibly non-matching set, K, of $\leq_0$ constraints. Question: Is there an algorithm that computes a matching substitution, S, for K and reports failure if the matching fails? The answer is "Yes". Many versions of this algorithm exist — we will be using the Algorithm $\mathcal{F}$ developed by Nielson, Nielson and Amtoft [15], based on ideas from Fuh and Mishra's work on type inference in the presence of subtypes [6]. The algorithm has been shown to be sound and complete: we will just mention the soundness property here.

Property 1 (Soundness of Algorithm $\mathcal{F}$) *Let K be a set of possibly non-matching $\leq_0$ constraints. Then Algorithm $\mathcal{F}$ computes a matching substitution for K, if one exists, and reports failure otherwise. More specifically, if a matching substitution for K exists, $\mathcal{F}(K)$ computes the triple, (S', K', C'), where S' is a matching substitution for K, **restricted to type variables only**,[9] K' is a set of **atomic** type constraints, and C' is a set of regular flow constraints, such that, $K', C' \vdash_{\mathsf{ST}} S'K$.*

In our inference algorithm, however, we will need to compute matching substitutions for a possibly non-matching set of $\leq_{2,1}$ constraints. The natural question is: Is there an algorithm, similar to Algorithm $\mathcal{F}$, that can compute a matching substitution for such a set? We will show that for every non-matching set of $\leq_{2,1}$ constraints *computed by the inference algorithm*, there exists a corresponding non-matching set of $\leq_0$ constraints. Thus, it suffices to use Algorithm $\mathcal{F}$ on this latter set. We formalise the notions below.

Consider the quadruple, $\langle K, C, P, S\rangle$, where K is a possibly non-matching set of $\leq_0$ constraints, C is a set of regular flow constraints, P is a possibly empty set of $\leq_{2,1}$ constraints of the form, $\{\tau_i \leq_{2,1} \sigma_i \mid i \in I\}$, and S is a substitution restricted to type variables only. Then $\langle K, C, P, S\rangle$ will be called a $\leq_{2,1}$ **matching problem**. The special case, $P = \emptyset$, will be called a $\leq_0$ **matching problem**.

Definition 6 (Solution to $\leq_{2,1}$ matching problems)

*A solution to the $\leq_{2,1}$ matching problem, $\langle K, C, P, S\rangle$, is a triple, (K', C', S'), where K' is a set of **atomic** type constraints, C' is a set of regular flow constraints, and S' is a matching substitution for K and is a matching substitution for the set, P — restricted to type variables only — such that, $K', C' \vdash_{\mathsf{ST}} S'K$, and $K', C' \vdash_{\mathsf{FL}} C$, and $K', C' \vdash_{\mathsf{ST}} S'P$.*

Note that in Definition 6, if $P = \emptyset$, then the $\leq_{2,1}$ matching problem reduces to the $\leq_0$ matching problem, $\langle K, C, \emptyset, S\rangle$, such that, if it has a solution, (K', C', S'), then, $K', C' \vdash_{\mathsf{ST}} S'SK$ and $K', C' \vdash_{\mathsf{FL}} C$. We also know that Algorithm $\mathcal{F}$ can compute a solution (if one exists) for the possibly non-matching set $S(K)$. Let $\mathcal{F}(S(K)) = (S'', K'', C'')$. Then, by soundness of $\mathcal{F}$, since $K'', C'' \vdash_{\mathsf{FL}} S''SK$ and $K'', C'' \cup C \vdash_{\mathsf{ST}} C$, we can assert that $(K'', C'' \cup C, S''S)$ is a solution to the $\leq_0$ matching problem, $\langle K, C, \emptyset, S\rangle$!

We will now show that any $\leq_{2,1}$ matching problem arising out of the inference algorithm can be rewritten into a $\leq_0$ matching problem. For the moment, the reader is requested to accept on faith that there can be only four basic forms of $\leq_{2,1}$ constraints arising from the inference algorithm:[10] (a)$t \leq_{2,1} \tau$, where τ is a $\mathbf{T_0}$ type, (b) $\tau \leq_{2,1} \tau_1 \wedge \tau_2$, (c) $(\tau_1 \xrightarrow{\xi} \tau_2) \leq_{2,1} t$, and (d) $(\tau_1 \xrightarrow{\xi} \tau_2) \leq_{2,1} (\sigma_1 \xrightarrow{\xi'} \sigma_2)$. The rewriting rules appear in Figure 3.

- $\langle K, C, P \cup \{t \leq_{2,1} \tau \mid \tau \in \mathbf{T_0}\}, S\rangle$
  $\Longrightarrow \langle S(K \cup \{t \leq_0 \tau\}), C, P, S\rangle$
- $\langle K, C, P \cup \{\tau \leq_{2,1} \tau_1 \wedge \tau_2\}, S\rangle$
  $\Longrightarrow \langle SK, C, P \cup \{\tau \leq_{2,1} \tau_1,\ \tau \leq_{2,1} \tau_2\}, S\rangle$
- $\langle K, C, P \cup \{(\tau_1 \xrightarrow{\xi} \tau_2) \leq_{2,1} t\}, S\rangle$
  $\Longrightarrow \langle K', C', P', S'\rangle$
  where t_1, t_2, ξ' *are fresh,* $K' = S'K$,
  $C' = C \cup \{\xi' \succeq \xi\}$,
  $P' = P \cup \{t_1 \leq_{2,1} \tau_1,\ \tau_2 \leq_{2,1} t_2\}$,
  $S' = [t \mapsto t_1 \xrightarrow{\xi'} t_2] \circ S$
- $\langle K, C, P \cup \{(\tau_1 \xrightarrow{\xi} \tau_2) \leq_{2,1} (\sigma_1 \xrightarrow{\xi'} \sigma_2)\}, S\rangle$
  $\Longrightarrow \langle SK, C', P', S\rangle$
  where $C' = C \cup \{\xi' \succeq \xi\}$,
  $P' = P \cup \{\sigma_1 \leq_{2,1} \tau_1,\ \tau_2 \leq_{2,1} \sigma_2\}$

Figure 3: Rewriting of $\leq_{2,1}$ constraints to $\leq_0$ constraints

Note that Figure 3 indeed defines, in rewrite rule form, an *algorithm* to transform $\leq_{2,1}$ constraints to $\leq_0$ constraints. The rewriting terminates, since every

[9] A substitution S is said to be restricted to type variables only, if for all type variables, t in its domain, $St = \tau$, where τ is a $\mathbf{T_0}$ type, and for all flow variables, ξ in its domain, $S\xi = \xi$.

[10] This will be clear by inspection when the inference algorithm is presented in the following section.

step of the rewriting creates a "smaller" set of $\leq_{2,1}$ constraints: by inspection, one of two things happens at every rule — either the number of type constructors ($\rightarrow$ or $\wedge$) decreases, or the number of $\leq_{2,1}$ inequalities is reduced. As a result, we are guaranteed that the $\leq_{2,1}$ matching problem, $\langle K, C, P, S\rangle$ will be eventually rewritten into the $\leq_{2,1}$ matching problem, $\langle K', C', \emptyset, S'\rangle$.

The following fact is immediate:

Fact 3 (Properties of the relation $\Longrightarrow$)
If $\langle K, C, P, S\rangle \Longrightarrow \langle K', C', P', S'\rangle$, then:

- *There exists a substitution, R, such that, $RS = S'$.*
- *$S'K' = K'$.*
- *$C' \supseteq C$ and all new flow constraints generated due to rewriting are between flow variables, i.e., of the form $\xi' \succeq \xi$.*
- *If (K'', C'', S'') is a solution to $\langle K', C', P', S'\rangle$, then, $(K'', C'', S''S')$ is a solution to $\langle K, C, P, S\rangle$.*

In summary, we have the following crucial theorem relating $\leq_{2,1}$ and $\leq_0$ matching problems.

Theorem 1
Let $\langle K, C, P, S\rangle$ be a $\leq_{2,1}$ matching problem, where P is a set of $\leq_{2,1}$ constraints obtained from the inference algorithm. Then, $\langle K, C, P, S\rangle \overset{+}{\Longrightarrow} \langle K', C', \emptyset, S'\rangle$. Furthermore, if $\mathcal{F}(K') = (S'', K'', C'')$, then, $(K'', C'' \cup C', S''S')$ is a solution to both $\langle K', C', \emptyset, S'\rangle$ and to $\langle K, C, P, S\rangle$.

4.2 The inference algorithm: an intuitive explanation

For a term e, the inference algorithm calculates the pair, $\langle TE,\ \tau\rangle$ — where TE is a $\mathbf{T_1}$ type environment, and τ is a $\mathbf{T_2}$ type — a set of type constraints, K, and a set of flow constraints, C. We will show in Section 5 that K is atomic, C is regular, and that all flows occurring in τ are *flow variables.* We will also need to demonstrate that $K, C, TE \vdash e : \tau$ is a valid derivation in $\mathbf{L_2}$.

We present the inference algorithm in inference-rule format in Figure 4. The algorithm is in a style reminiscent of Damas' Algorithm T [5]. The following notations are used:

Notation: TE_x denotes the type environment TE with all occurrences of x deleted; further, $(TE_1 + TE_2)(x) =$

$$\begin{cases} TE_1(x),\ x \in \mathbf{dom}(TE_1),\ \mathbf{x} \notin \mathbf{dom}(TE_2) \\ TE_2(x),\ x \in \mathbf{dom}(TE_2),\ \mathbf{x} \notin \mathbf{dom}(TE_1) \\ TE_1(x) \wedge TE_2(x),\ x \in \mathbf{dom}(TE_1) \text{ and} \\ x \in \mathbf{dom}(TE_2) \end{cases}$$

We will only focus on the rules for application since the ones for identifiers and λ-abstractions are easy.

There are two rules for inferring the type of an application, $e_0 e_1$. For the first, we need to show that $K'', C' \cup C'', S''S'(TE_0 + TE_1) \vdash e_0 e_1 : S''S'(t_2)$ is a valid derivation in $\mathbf{L_2}$. We can assume that the inference of e_0 yields the $\mathbf{T_1}$ type environment, TE_0, the $\mathbf{T_2}$ type, t, a set of atomic type constraints, K_0, and a set of regular flow constraints, C_0, so that, $K_0, C_0, TE_0 \vdash e_0 : t$ is a valid derivation in $\mathbf{L_2}$. In a similar manner, we can assume that $K_1, C_1, TE_1 \vdash e_1 : \tau$ is a valid derivation in $\mathbf{L_2}$.

$$\rhd x \ :\ \langle \{x : t\},\ t\rangle,\ \emptyset,\ \emptyset, \quad \textit{where } t \textit{ is fresh.}$$

$$\frac{\rhd e\ :\ \langle TE,\ \tau\rangle,\ K,\ C}{\rhd \lambda^\ell x.e\ :\ \langle TE,\ t \xrightarrow{\xi} \tau\rangle,\ K,\ C \cup \{\xi \succeq \{\ell\}\}}$$

$$\textit{where} \begin{cases} x \notin \mathbf{dom}(TE), \\ t,\ \xi \textit{ are fresh.} \end{cases}$$

$$\frac{\rhd e\ :\ \langle TE,\ \tau\rangle,\ K,\ C}{\rhd \lambda^\ell x.e\ :\ \langle TE_x,\ TE(x) \xrightarrow{\xi} \tau\rangle,\ K,\ C'}$$

$$\textit{where} \begin{cases} x \in \mathbf{dom}(TE), \\ \xi \textit{ is a fresh flow variable} \\ C' = C \cup \{\xi \succeq \{\ell\}\} \end{cases}$$

$$\frac{\rhd e_0\ :\ \langle TE_0,\ t\rangle,\ K_0,\ C_0 \qquad \rhd e_1\ :\ \langle TE_1,\ \tau\rangle,\ K_1,\ C_1}{\rhd e_0 e_1\ :\ S''S'\langle TE',\ t_2\rangle,\ K'',\ C' \cup C''}$$

$$\textit{where} \begin{cases} t_1,\ t_2,\ \xi \textit{ are fresh,} \\ K = K_0 \cup K_1,\ C = C_0 \cup C_1, \\ S = [t \mapsto t_1 \xrightarrow{\xi} t_2], \\ \langle SK, C, \{\tau \leq_{2,1} t_1\}, S\rangle \overset{+}{\Longrightarrow} \langle K', C', \emptyset, S'\rangle, \\ \mathcal{F}(K') = (S'', K'', C''), \textit{ and} \\ TE' = TE_0 + TE_1 \end{cases}$$

$$\frac{\rhd e_0\ :\ \langle TE_0,\ (\bigwedge_{i\in I} \sigma_i) \xrightarrow{\phi} \sigma\rangle,\ K_0,\ C_0 \qquad (\forall i \in I)\ \rhd e_1\ :\ \langle TE_i,\ \tau_i\rangle,\ K_i,\ C_i}{\rhd e_0 e_1\ :\ S''S'\langle TE',\ \sigma\rangle,\ K'',\ C' \cup C''}$$

$$\textit{where} \begin{cases} \forall i \in I, \textit{ the type and flow variables} \\ \textit{in } \langle TE_i,\ \tau_i\rangle,\ K_i \textit{ and } C_i, \textit{ are fresh,} \\ K = K_0 \cup \bigcup_{i\in I} K_i, \\ C = C_0 \cup \bigcup_{i\in I} C_i,\ P = \{\tau_i \leq_{2,1} \sigma_i \mid i \in I\}, \\ \langle K, C, P, [\,]\rangle \overset{+}{\Longrightarrow} \langle K', C', \emptyset, S'\rangle, \\ \mathcal{F}(K') = (S'', K'', C'') \textit{ and} \\ TE' = TE_0 + \sum_{i\in I} TE_i \end{cases}$$

Figure 4: Inference Algorithm for $\mathbf{L_2}$

Clearly, t ought to be a function type of the form, $t_1 \xrightarrow{\xi} t_2$, where t_1, t_2, ξ are fresh. Furthermore, τ, a $\mathbf{T_2}$ type, ought to be made a subtype of t_1, a $\mathbf{T_1}$ type. Let S be the substitution, $[t \mapsto t_1 \xrightarrow{\xi} t_2]$, $K = K_0 \cup K_1$, and $C = C_0 \cup C_1$. Note that while K is atomic, SK is a possibly non-matching set of $\leq_0$ type constraints. Similarly, the types τ and t_1 are possibly non-matching; hence to force a $\leq_{2,1}$ subtyping judgement between them, we must find a matching substitution, a set of atomic type constraints and a set of regular flow constraints, such that the obvious judgement can be asserted. Furthermore, the matching substitution should match the $\leq_0$ constraints in the set SK.

Now note that $\langle SK, C, \{\tau \leq_{2,1} t_1\}, S\rangle$ is a $\leq_{2,1}$ matching problem! Hence we can use the algorithm in Figure 3 to transform it into a $\leq_0$ matching problem, $\langle K', C', \emptyset, S'\rangle$. Algorithm $\mathcal{F}$ can compute a matching substitution for K', (if one exists), restricted to type variables only. By Theorem 1, we know that a solution for the $\leq_0$ problem also solves the $\leq_{2,1}$ problem, $\langle SK, C, \{\tau \leq_{2,1} t_1\}, S\rangle$. Accordingly, let $\mathcal{F}(K') = (S'', K'', C'')$. Then we can assert the subtyping judgement, $K'', C' \cup C'' \vdash_{\mathsf{ST}} S''S'(\tau) \leq_{2,1} S''S'(t_1)$. Furthermore, since $K'', C' \cup C'' \vdash_{\mathsf{ST}} S''S'K$, we can assert the judgements,
$K'', C' \cup C'', S''S'(TE_0 + TE_1) \vdash e_0 : S''S't_1 \xrightarrow{\xi} S''S't_2$, and
$K'', C' \cup C'', S''S'(TE_0 + TE_1) \vdash e_1 : S''S'\tau$.

Putting the pieces together, and using the rule for application and subtyping in $\mathbf{L_2}$, we obtain,
$K'', C' \cup C'', S''S'(TE_0 + TE_1) \vdash e_0 e_1 : S''S'(t_2)$.

In a similar fashion, we can show that for the second application rule, the derivation,
$K'', C' \cup C'', S''S'(TE_0 + \sum_{i \in I} TE_i) \vdash e_0 e_1 : S''S'(\sigma)$ is valid in $\mathbf{L_2}$.

5 Properties of the inference algorithm

We state without proof the following properties of the inference algorithm.

Lemma 1
For all expressions, e, if $\triangleright e : \langle TE, \tau\rangle, K, C$, then, K is atomic and C is regular. Furthermore, all flows in the inferred type, τ, are flow variables.

Theorem 2 (Soundness)
For all expressions, e, the inference algorithm always terminates either with failure,[11] *or, if $\triangleright e : \langle TE, \tau\rangle, K, C$, then, it is the case that $K, C, TE \vdash e : \tau$.*[12]

Notation: Say that $\langle TE, \tau\rangle \leq \langle TE', \tau'\rangle$, iff, $TE' \leq_1 TE$ and $\tau \leq_2 \tau'$. We then have the following theorem:

Theorem 3 (Completeness)
Suppose $K, C, TE \vdash e : \tau$. Then $\triangleright e : \langle TE', \tau'\rangle, K', C'$ succeeds, and there exists a matching substitution, R', of K', such that

(i) $K, C \vdash_{\mathsf{ST}} R'\langle TE', \tau'\rangle \leq \langle TE, \tau\rangle$.

(ii) $K, C \vdash_{\mathsf{ST}} R'K'$ *and* $K, C \vdash_{\mathsf{FL}} R'C'$.[13]

Given an expression, e, type inference either fails, or generates the pair, $\langle TE, \tau\rangle$, the set of atomic type constraints, K, and the set of regular flow constraints, C. Closure analysis now is just the solution of C, using the usual transitive closure algorithm.

6 Example

Consider the term, $(\lambda^1 z\,.\,zz)(\lambda^2 y\,.\,y)$. We can show that the inference algorithm applied to $\lambda^1 z\,.\,zz$ yields, $\langle \emptyset, [(t_3 \xrightarrow{\xi} t_4) \wedge t_1] \xrightarrow{\xi'} t_4\rangle, \{t_1 \leq_0 t_3\}, \{\xi' \succeq \{1\}\}$, and, applied to $\lambda^2 y\,.\,y$, yields $\langle \emptyset, t_5 \xrightarrow{\xi''} t_5\rangle, \emptyset, \{\xi'' \succeq \{2\}\}$. Now we can obtain the $\leq_{2,1}$ matching problem, $\langle K, C, P, S\rangle$, where:
$K = \{t_1 \leq_0 t_3\}$,
$C = \{\xi' \succeq \{1\}, \xi'' \succeq \{2\}, \bar{\xi''} \succeq \{2\}\}$,
$P = \{t_5 \xrightarrow{\xi''} t_5 \leq_{2,1} t_3 \xrightarrow{\xi} t_4, \bar{t_5} \xrightarrow{\bar{\xi''}} \bar{t_5} \leq_{2,1} t_1\}$,
and $S = [\ ]$. This matching problem can be reduced to the $\leq_0$ matching problem, $\langle K', C', \emptyset, S'\rangle$, where:
$K' = \{t_{11} \xrightarrow{\xi_{11}} t_{12} \leq_0 t_3, \bar{t_5} \leq_0 t_{12}, t_{11} \leq_0 \bar{t_5}, t_5 \leq_0 t_4, t_3 \leq_0 t_5\}$,
$C' = C \cup \{\xi_1 \succeq \bar{\xi''}\}$, $S' = [t_1 \mapsto t_{11} \xrightarrow{\xi_{11}} t_{12}]$.

Applying Algorithm $\mathcal{F}$ to K' yields the substitution: $[t_3 \mapsto t_{31} \xrightarrow{\xi_{31}} t_{32}, t_4 \mapsto t_{41} \xrightarrow{\xi_{41}} t_{42}, t_5 \mapsto t_{51} \xrightarrow{\xi_{51}} t_{52}]$, the set of atomic type constraints:
$\{t_{31} \leq_0 t_{11}, t_{12} \leq_0 t_{32}, \bar{t_5} \leq_0 t_{12}, t_{11} \leq_0 \bar{t_5}, t_{41} \leq_0 t_{51}, t_{52} \leq_0 t_{42}, t_{51} \leq_0 t_{31}, t_{32} \leq_0 t_{52}\}$, and the set of regular flow constraints,
$C'' = \{\xi_{31} \succeq \xi_{11}, \xi_{51} \succeq \xi_{31}, \xi_{41} \succeq \xi_{51}\}$.

Upon solving the set of constraints, $C' \cup C''$, we see that the type of the entire expression is $t_{41} \xrightarrow{\{2\}} t_{42}$, giving the expected result that it evaluates to the lambda labelled 2.

7 Discussion

A difference between the current framework and existing abstract interpretation-based approaches and Tang's effect systems-based approaches is that our analysis doesn't track function calls. Instead, we only calculate what set of functions every program point can possibly evaluate to. This automatically provides a call-tracking analysis: suppose a program point has type, $(t_1 \xrightarrow{\{2\}} t_2) \xrightarrow{\{1\}} (t_3 \xrightarrow{\{4\}} t_4)$, then we immediately know that it can evaluate to the function labelled 1 and the behaviour of 1 is that when applied, it will call the function 2 and yield the function 4. Moreover, we observe that 2 and 4 never call any functions.

The rules for type inference of an application, $e_0 e_1$, in Figure 4, reveals that the merging of the type environments in the consequent does *not* unify the possibly different types of a free variable occurring both in e and in e': rather the variable is given an intersection type. This is of crucial importance in providing polyvariance. Consider the example $(\lambda f.(\lambda x.fI)(f0))I$, where I is the identity combinator. Instead of giving a type $\top \to \top$ to f, and assigning the type $\top$ to the entire expression

[11] due to failure of matching in Algorithm $\mathcal{F}$.
[12] the proof uses the property of soundness of Algorithm $\mathcal{F}$.
[13] the proof uses the property of completeness of Algorithm $\mathcal{F}$.

(as is done in [17]) we can do better. First, let us annotate the term (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 and solving the set of flow constraints, we find that the type of the entire expression is $t_1 \xrightarrow{\{3\}} t_1$, showing that it evaluates to closure $\{3\}$. The type of the function part of the application becomes $[((t_1 \xrightarrow{\{3\}} t_1) \xrightarrow{\{4\}} (t_1 \xrightarrow{\{3\}} t_1)) \wedge (int \xrightarrow{\{4\}} int)] \xrightarrow{\{1\}} (t_1 \xrightarrow{\{3\}} t_1)$. which shows the expected polyvariance: the two uses of f expect the identity $\{4\}$; at one use point the identity calls the function 3 and returns it; at the other it expects an integer and returns an integer. Note that this automatically ensures that v is only going to be bound to the function labelled 3.

The impact of modularity is that at every stage of the inference algorithm, we can reduce the set of flow constraints and atomic type constraints, to some "normal form" and store it away. During linking, we will possibly generate some new constraints, but we do not need to recompute the "normal forms".

8 Conclusion and Future Work

We have developed a type-based closure analysis by a simple extension of the rank 2 intersection type system. The inherent polymorphism of intersection types is exploited to provide a polyvariant analysis. Type inference in the system is sound and complete and yields principal typings, resulting in a modular analysis.

Recursion has often been the bane of many type-based program analyses. Whereas we would very much like polymorphic recursion, type inference becomes undecidable. The type system in this paper can be easily extended to allow top-level universal quantification so that the following recursion rule suggested by Jim [10] can be added:

$$\frac{K, C, TE + \{x : \tau\} \vdash e : \sigma \qquad K, C \vdash_{\mathsf{ST}} \sigma \leq_{\forall_{2,1}} \tau}{K, C, TE \vdash \mu x.e : \sigma}$$

This would be an improvement on ML-style monomorphic recursion while still retaining principal typings and polyvariance. More interesting would be the extension of the framework to handle intersections for arbitrary depths for which, Jim hints, principal typings can be obtained.

A different approach would be to use Tofte and Talpin's region analysis [24], but this analysis, being for ML, uses the assumption of monovariant λ-bound variables, which results in loss of polyvariance if such a variable is used multiple times within the λ body. However, it remains to be seen how region-polymorphism for recursive definitions compares with the above rule for recursion.

For $\mathbf{T_0}$ types, our analysis can track the same calls as Tang's call-tracking analysis. Our analysis needs to be implemented, tested for precision, and analysed for complexity vis-a-vis its corresponding abstract interpretation-based analysis. It also remains to be seen how this work compares with the polymorphic splitting optimisation of Jagannathan and Wright [9].

We are continuing work on extending our framework with recursive types of the form $\mu t.\tau$, where τ is a $\mathbf{T_0}$ type. Such an extension would allow us to handle recursive data structures as well.

It will also be interesting to develop a generic, type-based program analysis framework that is modular and achieves precision comparable to that obtained via an abstract interpretation-based analysis. In particular, the precise relationship between flow information obtainable from a rank k type system and the k-CFA hierarchy proposed by Shivers [20] needs to be explored.

Acknowledgements: Thanks to Flemming Nielson and Hanne Riis Nielson for listening patiently to many presentations of this work, for their apposite comments, and for inviting me to work in their group. Thanks to the Church Project, in particular, to Assaf Kfoury and Bob Muller for discussions and encouragement, and to Jens Palsberg for his interest. For comments on drafts of this paper, I wish to thank Torben Amtoft, Olivier Danvy, Allyn Dimock, Pascal Fradet, Kirsten L. S. Gasser, Chris Hankin, Fritz Henglein, Thomas Jensen, Pierre Jouvelot, Andrew Kennedy, Sergey Kotov, Julia Lawall, Peter Ørbæk, Jakob Rehof, and David Schmidt.

References

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

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

[3] Torben Amtoft, Flemming Nielson, and Hanne Riis Nielson. Polymorphic subtyping for behaviour analysis. Unpublished manuscript, 1996.

[4] Patrick Cousot and Radhia Cousot. 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*, January 1977.

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

[6] Y. C. Fuh and Prateek Mishra. Type inference with subtypes. *Theoretical Computer Science*, 73(4):603–631, October 1990.

[7] Nevin Heintze. Control-flow analysis and type systems. In Alan Mycroft, editor, *Proceedings of the second International Static Analysis Symposium*, number 983 in Lecture Notes in Computer Science, pages 189–206, Glasgow, Scotland, September 1995.

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

[9] Suresh Jagannathan and Andrew Wright. Effective flow analysis for avoiding run-time checks. In Hanne Riis Nielson and Kirsten Lackner Solberg, editors, *Proceedings of Workshop on Types for Program Analysis*, DAIMI PB-495, pages 63–79. University of Aarhus, Denmark, 1995.

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

[11] Neil D. Jones. Flow analysis of lambda expressions. In *Proceedings of Eighth Colloquium on Automata, Languages, and Programming*, volume 115 of *LNCS*. Springer-Verlag, 1981.

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

[13] Christian Mossin. *Flow analysis of typed higher-order programs.* PhD thesis, DIKU, University of Copenhagen, January 1997.

[14] Flemming Nielson and Hanne Riis Nielson. Infinitary control flow analysis: a collecting semantics for closure analysis. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Principles of Programming Languages*, Paris, France, January 1997.

[15] Flemming Nielson, Hanne Riis Nielson, and Torben Amtoft. Polymorphic subtyping for effect analysis: the algorithm. In Mads Dam and Fredrik Orava, editors, *Proceedings of the fifth LOMAPS Workshop on Multiple-Agent Languages*, number 1192 in Lecture Notes in Computer Science, pages 207–243, 1997.

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

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

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

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

[20] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages or Taming Lambda.* PhD thesis, CMU, Pittsburgh, Pennsylvania, May 1991. Technical Report CMU-CS-91-145.

[21] Kirsten Lackner Solberg. *Annotated type systems for program analysis.* PhD thesis, Aarhus University, Aarhus, Denmark, 1995. DAIMI PB-498.

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

[23] Yan-Mei Tang. *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.

[24] Mads Tofte and Jean-Pierre Talpin. Implementation of the Typed Call-by-value λ-calculus using a Stack of Regions. In *Proceedings of the Twentyfirst Annual ACM Symposium on Principles of Programming Languages*, Portland, Oregon, January 1994.