First-Order Liveness Analysis via Type Inference

—DRAFT—

Juan Carlos Guzmán* Paul Hudak*

March 15, 1991

Abstract

In this paper we describe how to perform *liveness analysis* as a by-product of type inference in an extended Hindley-Milner type system. The analysis captures an approximation to the lifetime properties of objects, including polymorphic and recursive datatypes.

Our system relies on nested quantification over "liveness labels", which contrasts sharply with the Hindley-Milner system, in which quantification is restricted to the outermost level. Thus we cannot rely on a simple unification strategy to compute liveness of recursive functions and data structures. The system does have a "most-general-type" property, and we present an effective algorithm that computes such a type.

[1]Authors' address: Yale University
Department of Computer Science
P.O. Box 2158 Yale Station
New Haven, CT 06520
{guzman-juan,hudak-paul}@cs.yale.edu

1 Introduction

There has been considerable interest in recent years in using static type systems to specify constraints on and infer properties of program behaviors other than the normal notion of "type." Examples include Gifford, Lucassen, and Jouvelot's various uses of *effect systems* to capture mutability properties [LG88, GL86, JG89], Kuo and Mishra's type system that performs *strictness analysis* [KM89], Baker's simple extension to the Hindley-Milner type system to capture certain sharing properties [Bak90], and our own work on *single-threaded polymorphic lambda calculus* [GH90].

There are several motivations for using a static type system for these purposes instead of, for example, an abstract interpretation specification. First, the notation has certain advantages. In particular, the inference rules are specified in a modular way, with the inferencing algorithm specified independently. In contrast, denotational specifications generally rely on an understanding of domain theory, and the algorithm for inferring the properties is intrinsically tied to the inference rules.[1]

Second, we are interested not just in *inferring* program properties, but also in specifying certain *constraints*, or well-formedness criteria, on programs. A type system is the conventional method for specifying such static, context sensitive, constraints, and so is a natural choice. In addition, it allows the user to specify contraints explicitly by providing "type" signatures for functions, modules, etc.

Finally, for strongly-typed languages, the conventional type structure can be used as a "carrier" for whatever additional properties are being specified or inferred. In this context types have the following nice properties:

- They represent a *partition* of the domain of objects.
- They are *finite*.
- They reflect accurately the *structure* of objects (e.g., non-atomic and recursive types).

Thus properties that rely on notions of type structure can be specified as extensions to the conventional type structure, which simplifies the overall specification and aids reasoning about program correctness.

In this paper we describe how to perform *liveness analysis* as a by-product of type inference in an extended Hindley-Milner type system. The analysis captures an approximation to the lifetime properties of objects, including polymorphic and recursive datatypes. Although such information is of obvious use in a compiler (allowing, for example, certain forms of compile-time garbage collection), we also see its manifestation in a type system as a vehicle for designing languages whose static semantics depends upon some notion of object liveness.

[1] There has been some speculation that any abstract interpretation can be described within a type inference framework, but no general result of this sort is known.

For example, a language with "single-threaded" or "linear" types needs some notion of liveness on which to base the linearity (indeed, our result is a significant improvement over the liveness captured in our previous work [GH90]).

As an example of what our system is able to infer, consider the following function definitions:

$$\begin{aligned} \texttt{f(x,y)} &= \textit{if } x = y \textit{ then } x \textit{ else } x \\ \texttt{g(x,y)} &= \textit{if } x = y \textit{ then } y \textit{ else } y \\ \texttt{h(x,y)} &= \textit{if } x = y \textit{ then } x \textit{ else } y \end{aligned}$$

The conventional first-order Hindley-Milner type signature for each of these functions is the same (we assume the domain product operator binds tighter than the function operator):

$$\texttt{f,g,h} : \forall\alpha.\alpha \times \alpha \rightarrow \alpha$$

However, note that f discards its first argument and returns the second, g does the opposite, and h does one or the other depending on the equality of the two arguments. Thus our type system infers the following "types:"

$$\begin{aligned} \texttt{f} &:: \forall\alpha.\alpha^{\theta_1} \times \alpha^{\theta_2} \rightarrow \alpha^{\theta_1} \\ \texttt{g} &:: \forall\alpha.\alpha^{\theta_1} \times \alpha^{\theta_2} \rightarrow \alpha^{\theta_2} \\ \texttt{h} &:: \forall\alpha.\alpha^{\theta_1} \times \alpha^{\theta_2} \rightarrow \alpha^{\theta_1,\theta_2} \end{aligned}$$

The superscript on a type captures the liveness property. The way to read the liveness of f, for example, is: "f's first argument may persist as a result of applying f" or, alternatively, "f's result may contain references to its first argument." Similarly, for h: "h's result may contain references to either of its arguments."

As another example, the type of cons is:

$$\texttt{cons}: \forall\alpha.\alpha^{\theta_1} \times (\mathit{List}^{\theta_2}\ \alpha^{\theta_3}) \rightarrow (\mathit{List}^{\theta_2}\ \alpha^{\theta_1,\theta_3})$$

Note here that the type constructor *List* is annotated as well. We can interpret the above as: "The result returned from cons is a list which may reference part of the list structure of the second argument, and may contain elements that reference part of the first argument or elements of the second argument."

Aside from the obvious pragmatic advantages of our system, we have worked out some interesting theoretical difficulties relating to the type inference (i.e. reconstruction) problem. In particular, our system relies on nested quantification over liveness labels, which contrasts sharply with the Hindley-Milner system, in which quantification is restricted to the outermost level. Thus we cannot rely on a simple unification strategy to compute liveness of recursive functions and data structures. The system does have a "most-general-type" property, and we present an effective algorithm that computes such a type.

2 Liveness Analysis

We begin by defining the syntax of a simple functional language that will be used for all of our examples:

f	$\in$	$FIde$	function identifiers
v, x	$\in$	$VIde$	value identifiers
		$Ide = FIde + VIde$	identifiers
k	$\in$	Kon	constants
k_v	$\in$	Kon_v	constant values
k_f	$\in$	Kon_f	constants functions
e	$\in$	Exp	expressions
g	$\in$	$Funct$	functions

$$
\begin{array}{rcl}
\text{where } g & ::= & f \mid k_f \\
e & ::= & k_v \mid v \\
& & \mid g(e_1, \ldots, e_n) \\
& & \mid \mathit{let}\ v_1 = e_1; \\
& & \quad \ldots; \\
& & \quad v_m = e_m; \\
& & \quad f_1(x_{11}, \ldots, x_{1q_1}) = e'_1; \\
& & \quad \ldots; \\
& & \quad f_n(x_{n1}, \ldots, x_{nq_n}) = e'_n \\
& & \mathit{in}\ e
\end{array}
$$

Next we specify the syntax of type schemes, which is essentially that of Hindley-Milner [Hin78, Mil78, DM82], restricted to the first-order case:

$\sigma^v \in TypeSch$	$::=$	τ	Value Type Scheme
		$\mid \forall\alpha.\sigma^v$	Universal Quantification
$\sigma^f \in TypeSch$	$::=$	ϕ	Functional Type Scheme
		$\mid \forall\alpha.\sigma^f$	Universal Quantification
$\tau \in Type^v$	$::=$	$Int \mid Bool \mid \ldots$	Basic Types
		$\mid Pair\ \tau_1\ \tau_2$	Pairs
		$\mid List\ \tau$	Lists
		$\mid \alpha$	Variables
$\phi \in Type^f$	$::=$	$\mid \tau_1 \times \ldots \times \tau_n \rightarrow \tau_0$	Functions

We also assume the presence of the constants shown in Figure 1, having their traditional semantics.

$$
\begin{array}{rl}
\texttt{if}: & \forall\alpha.\mathit{Bool} \times \alpha \times \alpha \rightarrow \alpha \\
\texttt{null?}: & \forall\alpha.(\mathit{List}\ \alpha) \rightarrow \mathit{Bool} \\
\texttt{nil}: & \forall\alpha.(\mathit{List}\ \alpha) \\
\texttt{cons}: & \forall\alpha.\alpha \times (\mathit{List}\ \alpha) \rightarrow (\mathit{List}\ \alpha) \\
\texttt{head}: & \forall\alpha.(\mathit{List}\ \alpha) \rightarrow \alpha \\
\texttt{tail}: & \forall\alpha.(\mathit{List}\ \alpha) \rightarrow (\mathit{List}\ \alpha) \\
\texttt{pair}: & \forall\alpha\beta.\alpha \times \beta \rightarrow (\mathit{Pair}\ \alpha\ \beta) \\
\texttt{fst}: & \forall\alpha\beta.(\mathit{Pair}\ \alpha\ \beta) \rightarrow \alpha \\
\texttt{snd}: & \forall\alpha\beta.(\mathit{Pair}\ \alpha\ \beta) \rightarrow \beta
\end{array}
$$

Figure 1: Standard Type Signatures for Selected Constants

3 Syntax of Extended Types

In this section we extend our simple language with syntax to support liveness properties. Although verbose upon initial inspection, we will soon adopt some conventions to simplify the notation.

$\hat{\sigma} \in \mathit{TypeSch}_{Ext} ::=$	$\hat{\sigma}^v \mid \hat{\sigma}^f$	Extended Type Schemes
	$\hat{\chi}^v$	Extended Value Type Scheme
	$\mid \forall\alpha.\hat{\sigma}^v$	
$\hat{\sigma}^v \in \mathit{TypeSch}^v_{Ext} ::=$	$\hat{\chi}^v$	Extended Value Type Scheme
	$\mid \forall\alpha.\hat{\sigma}^v$	
$\hat{\chi}^v \in \mathit{LivenessSch}^v_{Ext} ::=$	$\hat{\tau}$	
	$\forall\theta.\hat{\chi}^v$	
$\hat{\sigma}^f \in \mathit{TypeSch}^f_{Ext} ::=$	$\hat{\chi}^f$	Extended Functional Type Scheme
	$\mid \forall\alpha.\hat{\sigma}^f$	
$\hat{\chi}^f \in \mathit{LivenessSch}^f_{Ext} ::=$	$\hat{\phi}$	
	$\forall\theta.\hat{\chi}^f$	
$\hat{\tau} \in \mathit{ObjType}_{Ext} ::=$	$\mathit{Int}^\delta \mid \mathit{Bool}^\delta \mid \ldots$	Basic Types
	$\mid (\mathit{Pair}\ \hat{\tau}_1\ \hat{\tau}_2)^\delta$	Pairs
	$\mid (\mathit{List}\ \hat{\tau})^\delta$	Lists
	$\mid \alpha^\delta$	Variables
$\hat{\phi} \in \mathit{FuncType}_{Ext} ::=$	$\hat{\tau}_1 \times \ldots \times \hat{\tau}_n \rightarrow \hat{\chi}^v$	Function Arguments ($n \geq 1$)
	$\mid \forall\theta.\hat{\phi}$	
$\delta \in \mathit{LivenessExp} ::=$	$\theta_1, \ldots, \theta_n$	Liveness Label Sequence ($n \geq 1$)

As should be clear from the examples given in the introduction, liveness information is expressed by annotating each type expression with a "liveness expression" ($\mathit{LivenessExp}$). The θ's are called *liveness labels*, and should not be confused with *type variables*; indeed,

they are not variables at all. The $\forall$ quantifier binds liveness labels, and is only permitted at the functional type level (informally, the liveness information of the argument is 'introduced' by universal quantification), and in the outermost level of the range of a function. The latter kind of quantification is found, for example, in the function `pair` which creates a new pair each time it is called.

Intuitively, liveness labels model the different "regions," or "pools" where objects can "live." A region can only host objects of one type. If the type is atomic (as in the case of Int^{θ}) objects are in the region (θ in this case). If the type is structured, like $(Pair\ Int^{\theta_1}\ Bool^{\theta_2})^{\theta_3}$, then only the "skeleton" of the type is in the region (in this case the pair cell), but the objects are not (in this case, the integer is in region θ_2, and the boolean in θ_3). Since regions are monomophic, θ_1, θ_2, and θ_3 must all be different. A type labeled with several labels, as in Int^{θ_1,θ_2}, indicates that the object may belong to either region.

A polymorphic functional type, such as $\alpha^{\theta} \rightarrow \alpha^{\theta}$, must expect its argument to be from any region, and thus we quantify over the labels of its arguments, as in:

$$\forall\theta.\alpha^{\theta} \rightarrow \alpha^{\theta}$$

with the meaning that the 'region' the result belongs to is the same as the argument's. A type such as:

$$\forall\theta_1.\alpha^{\theta_1} \rightarrow \forall\theta_2.\alpha^{\theta_2}$$

means that the result will be in an unspecified region, which cannot be assumed to be the same as the argument's. This is why we need nested quantification.

From the above specification of extended types, the functions `f`, `g`, and `h` shown in the introduction have the following types:

$$\begin{aligned}
\texttt{f} &:: \forall\alpha.\forall\theta_1\theta_2.\alpha^{\theta_1} \times \alpha^{\theta_2} \rightarrow \alpha^{\theta_1} \\
\texttt{g} &:: \forall\alpha.\forall\theta_1\theta_2.\alpha^{\theta_1} \times \alpha^{\theta_2} \rightarrow \alpha^{\theta_2} \\
\texttt{h} &:: \forall\alpha.\forall\theta_1\theta_2.\alpha^{\theta_1} \times \alpha^{\theta_2} \rightarrow \alpha^{\theta_1\theta_2}
\end{aligned}$$

3.1 Syntactic Conventions

It is quite common to have many different liveness labels within a type expression – technically, each and every node is entitled to one – and each has an associated point at which it is quantified. But it provides useful information only when it appears in several places, thus capturing a relationship between those places. Therefore to simplify the notation we adopt the following conventions:

1. η will be used instead of θ when θ is bound in the innermost $\hat{\chi}$ (either $\hat{\chi}^v$, or $\hat{\chi}^l$) that encloses all ocurrences of θ, and the binding of η will be omitted. As a result, the type for the example function `f` given in the introduction will be written as:

   $$\texttt{f} :: \forall\alpha.\alpha^{\eta_1} \times \alpha^{\eta_2} \rightarrow \alpha^{\eta_1}$$

   Further, we omit liveness variables that:

$$
\begin{array}{rl}
\texttt{if}: & \forall\alpha.Bool \rightarrow \alpha^{\theta_1} \rightarrow \alpha^{\theta_2} \rightarrow \alpha^{\theta_1\,\theta_2} \\
\texttt{null?}: & \forall\alpha.(List\ \alpha) \rightarrow Bool \\
\texttt{nil}: & \forall\alpha.(List\ \alpha) \\
\texttt{cons}: & \forall\alpha.\alpha^{\theta_1} \rightarrow (List^{\theta_2}\ \alpha^{\theta_3}) \rightarrow (List^{\theta_2}\ \alpha^{\theta_1,\theta_3}) \\
\texttt{head}: & \forall\alpha.(List\ \alpha^{\theta}) \rightarrow \alpha^{\theta} \\
\texttt{tail}: & \forall\alpha.(List^{\theta_1}\ \alpha^{\theta_2}) \rightarrow (List^{\theta_1}\ \alpha^{\theta_2}) \\
\texttt{pair}: & \forall\alpha\beta.\alpha^{\theta_1} \rightarrow \beta^{\theta_2} \rightarrow (Pair\ \alpha^{\theta_1}\ \beta^{\theta_2}) \\
\texttt{fst}: & \forall\alpha\beta.(Pair\ \alpha^{\theta}\ \beta) \rightarrow \alpha^{\theta} \\
\texttt{snd}: & \forall\alpha\beta.(Pair\ \alpha\ \beta^{\theta}) \rightarrow \alpha^{\theta}
\end{array}
$$

Figure 2: Extended Types for Constants

(a) are bound according to this convention, and

(b) are used only once in the type expression, and it is the only label annotating the type node (i.e., is not part of a l.u.b.).

The type of `f` is then further simplified to

$$\texttt{f} :: \forall\alpha.\alpha^{\eta_1} \times \alpha \rightarrow \alpha^{\eta_1}$$

2. We will write $(List^{\theta}\ \hat{\tau})$ and $(Pair^{\theta}\ \hat{\tau}_1\ \hat{\tau}_2)$ instead of $(List\ \hat{\tau})^{\theta}$ and $(Pair\ \hat{\tau}_1\ \hat{\tau}_2)^{\theta}$, respectively. Following this convention, the extended type of `tail` is written as

$$\texttt{tail}: \forall\alpha.(List^{\theta_1}\ \alpha^{\theta_2}) \times (List^{\theta_1}\ \alpha^{\theta_2})$$

3. We will write $\{\theta_1,\ldots,\theta_n\}\hat{\chi}$ when we need to make explicit that $\hat{\chi}$ has free liveness labels θ_1 through θ_n. Sometimes, we will write $\Theta\hat{\chi}$, if the particular variables are not important, but only the fact that some free labels exists.

4. We will write θ^{α} instead of θ when we need to stress the fact that label θ always qualifies the type variable α.

Following the conventions, the types for `f`, `g`, and `h` specified above are:

$$
\begin{array}{ll}
\texttt{f} :: & \forall\alpha.\alpha^{\eta_1} \times \alpha \rightarrow \alpha^{\eta_1} \\
\texttt{g} :: & \forall\alpha.\alpha \times \alpha^{\eta_2} \rightarrow \alpha^{\eta_2} \\
\texttt{h} :: & \forall\alpha.\alpha^{\eta_1} \times \alpha^{\eta_2} \rightarrow \alpha^{\eta_1,\eta_2}
\end{array}
$$

Figure 2 show the type signatures for the assumed constants, using the above conventions. Note that when all liveness information is removed, the resulting type is exactly the standard type for the constant.

3.2 Well-Formed Extended Types

In order to keep the distinction of regions to a minimun, we note that if a 'new' region is always associated with a pre-existing one, then it is not necessary to distinguish them. Consider cons, for example: it takes an object and a list, and creates a new cell containing the object and a pointer to the list. Its extended type could thus be:

$$\texttt{cons}: \forall\alpha.\alpha^{\eta_1} \times (List^{\eta_2}\ \alpha^{\eta_3}) \rightarrow (List^{\eta_2,\eta_4}\ \alpha^{\eta_1,\eta_3})$$

but the fact that η_4 is a 'new region', suggests that we can assume the region η_2 has yet another cell, thus simplifying the type somewhat to:

$$\texttt{cons}: \forall\alpha.\alpha^{\eta_1} \times (List^{\eta_2}\ \alpha^{\eta_3}) \rightarrow (List^{\eta_2}\ \alpha^{\eta_1,\eta_3})$$

Another characteristic of functional types is that given that its argument can belong to any region, and has no restrictions other than those implied by its type,

- all free labels of the argument must appear only once there (i.e., the argument must be *linear* on all free labels), and
- all free labels of the argument must be universally quantified in the function's type.

Well-formed extended types must conform to these restrictions, as well as the additional restriction that all instances of the same label must be attached to identical type schemes. In particular, if an instance of a label is attached to a type variable, then all its instances must be attached to the same type variable.

The following type is well-formed:

$$\forall\alpha.\alpha^{\eta_1} \times (List^{\eta_2}\ \alpha^{\eta_3}) \rightarrow (List^{\eta_2}\ \alpha^{\eta_1,\eta_3})$$

But the following are not:

$$\forall\alpha.\alpha^{\eta_1} \times (List^{\eta_2}\ \alpha^{\eta_1}) \rightarrow (List^{\eta_2}\ \alpha^{\eta_1})$$
$$\forall\alpha.\alpha^{\eta_1} \times (List^{\eta_2}\ \alpha^{\eta_3}) \rightarrow (List^{\eta_2}\ \alpha^{\eta_1,\eta_2,\eta_3})$$
$$\forall\alpha\beta.(List^{\eta}\ \alpha) \rightarrow (List^{\eta}\ \beta)$$

The first type is ill-formed because η_1 is used non-linearly in the argument (once in the first, and once in the second argument), the second one is ill-typed because η_2 labels a list node in one instance, and a variable in another, and the third one is not valid because η is labelling lists of possibly different type.

For the remainder of the paper, we will consider only well-formed extended types.

3.3 Operations on Extended Types

3.3.1 Substitution

Substitution on extended types is an extension of substitution on standard types. The major problem on the extended type language is that substitution on types implies some substitution on the liveness labels associated, so that the pairing between types and labels is maintained. The actual definitions will appear in the final paper in the appendix. In this section, we will provide only an intuitive explanation on how they work. Suppose we have the extended type for g (as in the introduction)

$$\hat{\tau} = \alpha^{\eta_1} \times \alpha^{\eta_2} \rightarrow \alpha^{\eta_1 \sqcup \eta_2}$$

and want to do the substitution (on types) (i.e., suppose the call g (Pair x y) is found, where x and y have type α)

$$\tau[(Pair\ \alpha_1\ \alpha_2)/\alpha]$$

This substitution imposes 'structural restrictions' on η_1, and η_2—these regions will now be subdivided into three parts, one for the Pair, skeleton, and one for each component of the pair. This implies that η_1, and η_2 need to be replaced by other labels, which will annotate the pairs. These labels need to be bound in the same place where the original ones were. The labels that replace η_1 must be different from those that replace η_2 (in order not to introduce additional constraints in the type), and α's that are annotated by both labels, need to be replaced by types that are annotated by the conjunction of the corresponding labels that replace η_1, and η_2—this can be achieved by *zipping* the replacements of α^{η_1}, and α^{η_2}. The following type results after the replacement:

$$\hat{\tau} = (Pair^{\eta_{11}}\ \alpha^{\eta_{12}}\ \alpha^{\eta_{13}}) \times (Pair^{\eta_{21}}\ \alpha^{\eta_{22}}\ \alpha^{\eta_{23}}) \rightarrow (Pair^{\eta_{11},\eta_{21}}\ \alpha^{\eta_{12},\eta_{22}}\ \alpha^{\eta_{13},\eta_{23}})$$

Substitution on liveness labels on the contrary is done in the standard way, provided the labels involved in the operation are *compatible* (they annotate the same type, or are newly introduced labels). E.g.,

$$(Pair^{\theta_1}\ \alpha^{\theta_2}\ \alpha^{\theta_3})[\theta_2/\theta_3] \equiv (Pair^{\theta_1}\ \alpha^{\theta_2}\ \alpha^{\theta_2})$$

3.3.2 Instance Relation

We are interested in building a partial order on types, using the substitution relation. Intuitively, one type is an instance of another if it can be obtained from the latter by applying a suitable substitution. Alternatively, it is also said that the latter type is 'more general' than the former one.

1. Parameter Type Instance

$$\hat{\sigma} \succeq \hat{\sigma}[\tau/\alpha]$$

2. Generic Type Instance

$$\forall\alpha.\hat{\chi} \succeq \forall\alpha'_1 \ldots \alpha'_n.\hat{\chi}[\hat{\tau}/\alpha]$$

when α_i are not free in $\forall\alpha.\hat{\chi}$.

3. Parameter Liveness Label Instance

$$\hat{\sigma} \succeq \hat{\sigma}[\delta/\theta]$$

4. Generic Liveness Label Instance on χ

$$\forall\hat{\theta}.\hat{\chi} \succeq \forall\theta_1 \ldots \theta_n.\hat{\chi}[\delta/\theta]$$

when θ_i are not free in $\forall\theta.\hat{\chi}$.

Relations 1, 2, and 3 are extended to work on all different categories of extended types ($\hat{\chi}$, and $\hat{\tau}$). Note that even though types are not shallow with respect to liveness labels, only outermost labels can be instantiated using rule 4. This is due to the lack of rules that would allow relating expresssions by using the relation of their constituents.

3.4 Extended-Type Inference Rules

The functions $\mathcal{K}^f$, and $\mathcal{K}^v$ return the extended type for constant functional, and value constants. Alternatively, these extended types can be encoded in the initial environment. An assertion of the form $T \vdash e : \hat{\chi}^v$ expresses the fact that under assumptions T, e has the extended type χ^v. The assertion $T \overset{f}{\vdash} g : \hat{\chi}^f$ expresses a similar fact but relating functions to their extended type. The inference rules are very similar to a typical Hindley-Milner type system, therefore we will restrict our comments on issues relevant to the liveness analysis.

Function Designators

1. Functional Constants

$$A \overset{f}{\vdash} k_f : (\mathcal{K}^f\ k_f)$$

The extended type of a functional constant looked up in $\mathcal{K}^f$.

2. Functional Identifiers

$$T \overset{f}{\vdash} f : A(f)$$

The extended type of a functional identifier is retrieved from the environment T.

Value Expressions

3. Value Constants

$$T \vdash k_v : (\mathcal{K}_v\ k_v)$$

4. Value Identifiers

$$T \vdash v : A(v)$$

5. Applications

$$\frac{\begin{array}{c} T \overset{f}{\vdash} g : (\forall\theta_{11}\ldots\theta_{1m_1}\ldots\theta_{n1}\ldots\theta_{nm_n}.\hat{\tau}_1 \times \ldots \times \hat{\tau}_n \to \hat{\chi}) \\ T \vdash e_1 : \hat{\tau}'_1 \qquad \ldots \qquad e_n : \hat{\tau}'_n \\ \hat{\tau}'_i = \hat{\tau}[\delta_{i1}/\theta_{i1}, \ldots, \delta_{im_i}/\theta_{im_i}] \qquad i = 1..n \end{array}}{T \vdash g(e_1, \ldots, e_n) : \hat{\chi}[\delta_{11}/\theta_{11}, \ldots, \delta_{nm_n}/\theta_{nm_n}]}$$

For technical reasons regarding well-formedness of extended-types, it is required that $\hat{\tau}'_i$ be a liveness-instance of $\hat{\tau}_i$. This is a departure from the Hindley-Milner inference rules, which require τ_i, and τ'_i to be the same.

1. Let expression

$$\frac{\begin{array}{c} T' = T\left[\begin{array}{c} v_1 \mapsto \hat{\chi}_1, \ldots, v_m \mapsto \hat{\chi}_m, \\ f_1 \mapsto \forall\theta_{11}\ldots\theta_{1p_1}.\hat{\tau}_{11} \times \ldots \times \hat{\tau}_{1q_1} \to \hat{\chi}'_1, \\ \ldots, \\ f_n \mapsto \forall\theta_{n1}\ldots\theta_{np_n}.\hat{\tau}_{n1} \times \ldots \times \hat{\tau}_{nq_n} \to \hat{\chi}'_n \end{array}\right] \\ T' \vdash e_1 : \hat{\chi}_1, \ldots, e_m : \hat{\chi}_m \\ T'[x_{11} \mapsto \hat{\tau}_{11}, \ldots, x_1q_1 \mapsto \hat{\tau}_{1q_1}] \overset{f}{\vdash} e'_1 : \hat{\chi}'_1 \\ \ldots \\ T'[x_{n1} \mapsto \hat{\tau}_{n1}, \ldots, x_nq_n \mapsto \hat{\tau}_{nq_n}] \overset{f}{\vdash} e'_n : \hat{\chi}'_n \\ T' \vdash e : \hat{\tau} \end{array}}{T' \vdash \left(\begin{array}{l} let\ v_1 = e_1; \\ \quad \ldots; \\ \quad v_m = e_m; \\ \quad f_1(x_{11}, \ldots, x_{1q_1}) = e'_1; \\ \quad \ldots; \\ \quad f_n(x_{n1}, \ldots, x_{nq_n}) = e'_n \\ in\ e \end{array}\right) : \hat{\tau}}$$

This rule, although complex in appearance, is a direct extension to the *let*-rule in the Hindley-Milner type system. Note that polymorphism is allowed in the type of the bound identifiers.

2. Generalization

$$\frac{T \vdash e : \hat{\sigma} \qquad \alpha \text{ not free in } T}{T \vdash e : \forall\alpha.\hat{\sigma}}$$

$$\frac{T \vdash e : \hat{\chi} \quad \theta \text{ not free in } T}{T \vdash e : \forall\theta.\hat{\chi}}$$

3. Instantiation

$$\frac{T \vdash e : \hat{\chi} \quad \hat{\chi} \succeq \hat{\chi}'}{T \vdash e : \hat{\chi}'}$$

$$\frac{T \vdash e : \hat{\sigma} \quad \hat{\sigma} \succeq \hat{\sigma}'}{T \vdash e : \hat{\sigma}'}$$

Note that having the $\forall$ quantifier arbitrarily nested implies that expressions using this quantifier potentially generate a different instance every time the quantifier gets removed, i.e., every time the expression appears in the program text. This is reminiscent of the polymorphism introduced in let-expressions in a Hindley-Milner type system) where each time a let-bound value gets used, a 'fresh' instantiation of its type is manipulated.

4 Type Reconstruction

In this section we present how extended types can be effectively computed. Even though there exists algorithms that directly compute the 'extended type' of expressions, we have chosen to show an algorithm that merely 'extends the type' of a pre-inferred type. Such an algorithm has the following advantages:

- it works exclusively on liveness properties, and not on the basic type; this is a modular design in the same manner that lexer and parser are modular in compilation.
- No need for unification, just pattern matching, so, no need to to carry substitution environments around.

4.1 Overview

This algorithm departs from traditional type inferring algorithms in that this does not work 'top-down', but rather recognizes the dependencies among different identifiers in the program and always works on those identifiers for which all their dependencies have been solved. The following is an overview of the strategy used:

1. α-rename the program in such a way to avoid let-bound identifier name clashes.[2]

2. Build a *dependency graph* of the let-bound identifiers in the program as follows: if identifier x depends on identifier y for its value (a reference to y appears inside x's body), then put a dependence arc from x to y.

3. Collapse each group of nodes with cyclic dependencies in the graph into a *collection*, so that if x is dependent on y and vice versa, then x and y are both in the same collection. The resulting graph is a DAG.

4. Order the collections topologically, beginning with ones that have no outgoing edges (except, perhaps, to themselves).

5. Infer the type for nodes in collections in the order given by the topological sort. Each collection is to be solved using the technique given below.

4.1.1 Solving Each Collection

Identifiers that belong to the same collection are interdependent—the type of either of them may affect the value of the rest. Thus, we must solve all of them at the same time. Further, for collections bigger than one, the collection is recursive, thus we need to use fixpoint finding techniques to assign values to the different identifiers of the collection. We will do so by using a standard ascending Kleene chain technique.

The initial extended type of each element in the collection will be linear in all its liveness labels. E.g., if the type v is $(Pair\ Int\ Int)$, then its initial extended type will be $\forall\theta_1\theta_2\theta_3.Pair^{\theta_1}\ Int^{\theta_2}\ Int^{\theta_3}$; if f is of type $Int \rightarrow Int$, then its initial extended type will be $\forall\theta_1.Int^{\theta_1} \rightarrow \forall\theta_2.Int^{\theta_2}$. This represents the least extended type assignable to the identifier, since all other extended types are instances of this one.

Compute a new extended type for each identifier in the collection by assuming the current types and using algorithms $\mathcal{W}^f$, and $\mathcal{W}^v$. This is iterated until all the computed values match the assumed ones (modulo α-renaming).

4.2 Algorithm

Preliminaries

One of the benefits of the α-conversion performed in the first step of the algorithm is that we can associate to the program a *flat* environment containing the Hindley-Milner, and extended

[2]This is more a matter of convenience, since this way, a name uniquely identifies a program point, and thus also acts as a label.

types for every let-bound identifier. By abuse of notation we annotate the identifiers with their types rather than keeping an explicit environment.

$$\begin{aligned} T \in TEnv_{Ext} : &\ Ide \rightarrow TypeSch_{Ext} \\ e \in AExp : &\ Exp \times TEnv_{Ext} \end{aligned}$$

Given an annotated program, and a let-bound identifier, we obtain the definition corresponding to the identifier within the program using the function *definition*, we also retrieve the (extended) type of an identifier by using the function *typeof*, and we build a Damas-and-Milner-like assumption environment at the point the identifier is bound with the function *assumption-env*:

$$\begin{aligned} \mathit{definition} : &\ AExp \rightarrow Ide \rightarrow AExp \\ \mathit{typeof} : &\ AExp \rightarrow Ide \rightarrow TypeSch_{Ext} \\ \mathit{assumption\text{-}env} : &\ AExp \rightarrow Ide \rightarrow TEnv_{Ext} \end{aligned}$$

We use the function *ext* to build an initial "extended type" corresponding to a Hindley-Milner type as exposed in the section 4.1.1. Additionally, we use $inst_{ll}$, and $inst_{lv}$ to instantiate type variables, and outermost liveness labels; i.e., they act as universal quantifier eliminators. Their signatures are:

$$\begin{aligned} \mathit{ext} : &\ TypeSch \rightarrow TypeSch_{Ext} \\ inst_{ll}, inst_{lv} : &\ TypeSch_{Ext} \rightarrow TypeSch_{Ext} \end{aligned}$$

Finally, we use the 'closure' of an (extended) type $\hat{\tau}$ with respect to assumptions T, as

$$\overline{T}(\hat{\tau}) = \forall \theta_1 \ldots \theta_n . \hat{\tau}$$

where $\theta_1, \ldots, \theta_n$ are the liveness labels occurring free in $\hat{\tau}$, but not it T [DM82].

Functions identified with $\mathcal{T}$ implement the strategy for selection of nodes that was previously discussed, while those identified with $\mathcal{W}$ implement the actual extended type inference. The type signatures of all these are as follows[3]:

$$\begin{aligned} \mathcal{T}_p : &\ AExp \rightarrow AExp \\ \mathcal{T}^l : &\ FIde \rightarrow AExp \rightarrow TypeSch^l_{Ext} \\ \mathcal{T}^v : &\ VIde \rightarrow AExp \rightarrow TypeSch^v_{Ext} \\ \mathcal{T}_c : &\ 2^{VIde+FIde} \rightarrow AExp \rightarrow AExp \\ \mathcal{W}^l : &\ Funct \rightarrow Env \rightarrow TypeSch^l_{Ext} \\ \mathcal{W}^v : &\ Exp \rightarrow Env \rightarrow TypeSch^v_{Ext} \end{aligned}$$

[3]The operator *foldl* is a standard higher-order list operator of type

$$foldl :: (\beta \rightarrow \alpha \rightarrow \beta) \rightarrow \beta \rightarrow (List\ \alpha) \rightarrow \beta$$

defined as:

$$foldl\ f\ z\ [x_1, x_2, \ldots, x_n] = f\ (\ldots (f\ (f\ z\ x_1)\ x_2) \ldots)\ x_n$$

The reader is asked to read [BW88] for details

$$\begin{aligned}\mathcal{T}_p\ p\ =\ & let\ p' = \alpha\text{-}rename\ p \\ & \quad G = dependency\ p \\ & \quad G' = collapse\text{-}cycles\ G \\ & \quad cs = topological\text{-}sort\ G' \\ & in\ \ \mathcal{W}^v\ (foldl\ \mathcal{T}^c\ p\ cs)\end{aligned}$$

$$\begin{aligned}\mathcal{T}^f\ f\ p\ =\ & let\ (f(x_1^{\tau_1}, \ldots, x_n^{\tau_n}) = e) = definition\ f\ p \\ & \quad T = assumption\text{-}env\ f\ p \\ & \quad \hat{\tau}_1 = inst_{ll}\ (ext\ \tau_1) \\ & \quad \ldots \\ & \quad \hat{\tau}_n = inst_{ll}\ (ext\ \tau_n) \\ & \quad \hat{\tau} = \mathcal{W}^v\ e\ T[x_1 \mapsto \hat{\tau}_1, \ldots, x_n \mapsto \hat{\tau}_n] \\ & in\ simplify\ (\overline{T}\ ((\hat{\tau}_1 \times \ldots \times \hat{\tau}_n) \rightarrow \hat{\tau}))\end{aligned}$$

$$\begin{aligned}\mathcal{T}^v\ v\ p\ =\ & let\ (v = e) = definition\ f\ p \\ & \quad T = assumption\text{-}env\ e\ p \\ & in\ \ \hat{\tau} = \mathcal{W}^v\ e\ T\end{aligned}$$

$$\begin{aligned}\mathcal{T}^c\ \{v_1, .., v_m, f_1, \ldots, f_n\}\ p = & let\ \hat{\chi}_1^v = \mathcal{T}^v\ v_1\ p \\ & \quad \ldots \\ & \quad \hat{\chi}_n^f = \mathcal{T}^f\ f_n\ p \\ & in\ \ if\ (\hat{\chi}_1^v = typeof\ v_1\ p)\ and\ \ldots\ and\ (\hat{\chi}_n^f = typeof\ f_n\ p) \\ & \quad then\ p \\ & \quad else\ \mathcal{T}_c\ \{v_1, .., v_m, f_1, \ldots, f_n\}\ p[v_1 \mapsto \hat{\chi}_1^v, \ldots, f_n \mapsto \hat{\chi}_n^f]\end{aligned}$$

$$\begin{aligned}\mathcal{T}^c\ \{v_1, .., v_m, f_1, \ldots, f_n\}\ p = & let\ \hat{\chi}_1^v = typeof\ v_1 \\ & \quad \ldots \\ & \quad \hat{\chi}_n^f = typeof\ f_n \\ & \quad \ldots\ . \\ & in\ \ldots\end{aligned}$$

Function Designators

For function designators, we have the function $\mathcal{W}^f$ of the form

$$\begin{aligned}\mathcal{W}^f\ g\ T_i\ =\ & case\ g\ of \\ & \quad clauses\end{aligned}$$

where *clauses* are the following ones:

1. Functional Constants

$$
\begin{array}{l}
g = k_f^t \rightarrow \\
\quad let\ \hat{\chi}^f = Inst_{ll}\ Inst_{tv}\ (\mathcal{K}^f\ k_f) \\
\qquad S = \mathcal{PM}\ \hat{\chi}^f\ t \\
\quad in\ Subst\ \hat{\chi}^f\ S
\end{array}
$$

2. Functional Identifiers

$$
\begin{array}{l}
g = f^t \rightarrow \\
\quad let\ \hat{\chi}^f = Inst_{ll}\ Inst_{tv}\ (T_i\ f) \\
\qquad S = \mathcal{PM}\ \hat{\chi}^f\ t \\
\quad in\ Subst\ \hat{\chi}^f\ S
\end{array}
$$

Value Expressions

For value expressions, we also have the function $\mathcal{W}^v$ of a similar form as the above definition

$$
\begin{array}{lcl}
\mathcal{W}^v\ e\ T_i & = & case\ g\ of \\
 & & \quad clauses
\end{array}
$$

where *clauses* are the following ones:

1. Value Constants

$$
\begin{array}{l}
e = k_v^t \rightarrow \\
\quad let\ \hat{\chi}^v = Inst_{ll}\ Inst_{tv}\ (\mathcal{K}^v\ k_v) \\
\qquad S = \mathcal{PM}\ \hat{\chi}^v\ t \\
\quad in\ Subst\ \hat{\chi}^v\ S
\end{array}
$$

2. Value Identifiers

$$
\begin{array}{l}
e = v^t \rightarrow \\
\quad let\ \hat{\chi}^v = Inst_{ll}\ Inst_{tv}\ (T_i\ v) \\
\qquad S = \mathcal{PM}\ \hat{\chi}^v\ t \\
\quad in\ Subst\ \hat{\chi}^v\ S
\end{array}
$$

3. Applications

$$
\begin{array}{l}
e = g(e_1, \ldots, e_n) \rightarrow \\
\quad let\ (\forall \theta_{11} \ldots \theta_{1m_1} \ldots \theta_{n1} \ldots \theta_{nm_n}.\hat{\tau}_1 \times \ldots \times \hat{\tau}_n \rightarrow \hat{\chi}^v) = \mathcal{W}^f\ g\ T_i \\
\qquad \tau_1[\theta'_{11}/\theta_{11}, \ldots, \theta'_{1m_1}/\theta_{1m_1}] = \mathcal{W}^v\ e_1\ T_i \\
\qquad \ldots \\
\qquad \tau_n[\theta'_{n1}/\theta_{n1}, \ldots, \theta'_{nm_n}/\theta_{nm_n}] = \mathcal{W}^v\ e_n\ T_i \\
\quad in\ S\ \hat{\chi}^v[\theta'_{11}/\theta_{11}, \ldots, \theta'_{1m_1}/\theta_{1m_1}, \ldots, \theta'_{n1}/\theta_{n1}, \ldots, \theta'_{nm_n}/\theta_{nm_n}]
\end{array}
$$

4. Let Expression

$$e = \left(\begin{array}{l} let\ v_1^{\chi_1^v} = e_1; \\ \quad \ldots; \\ \quad v_m^{\chi_m^v} = e_m; \\ \quad f_1^{\chi_1^f}(x_{11}, \ldots, x_{1q_1}) = e_1'; \\ \quad \ldots; \\ \quad f_n^{\chi_n^f}(x_{n1}, \ldots, x_{nq_n}) = e_n' \\ in\ e \end{array} \right) \rightarrow$$

$$\mathcal{W}^v\ e\ T_i[v_1 \mapsto \chi_1^v, \ldots, v_m \mapsto \chi_m^v, f_1 \mapsto \chi_1^f, \ldots, f_n \mapsto \chi_n^f]$$

The extended type of all identifiers needed to compute the type of e are assumed to be already computed. This includes identifiers bound in the construct. This is consequence of performing the topological sort of nodes at function $\mathcal{T}_p$.

4.2.1 Example:

Consider the following program:

```
as = (Cons 1 (Cons 2 (Cons 3 Nil)))
append(xs,ys) = if (xs=Nil)
                  then ys
                  else Cons(head(xs),append(tail(xs),ys))
reverse(xs) = if (xs=Nil)
                then Nil
                else append(reverse(tail(xs)),Cons(head(xs),Nil))
in reverse(as)
```

The standard types of all let-bound identifiers of the preceeding program are:

$$\begin{array}{rc} \texttt{as}: & List\ Int \\ \texttt{append}: & \forall\alpha.(List\ \alpha) \times (List\ \alpha) \rightarrow (List\ \alpha) \\ \texttt{reverse}: & \forall\alpha.(List\ \alpha) \rightarrow (List\ \alpha) \end{array}$$

The initial extended type assignment for them is exactly as above, but interpreted as extended types. A topological order of the identifiers is {as, append, reverse}. Computing the extended type following this order leaves the type for as unchanged, the type for append converges after one iteration, and so does the type for reverse. The extended types are:

$$\begin{array}{rc} \texttt{as}: & List\ Int \\ \texttt{append}: & \forall\alpha.\forall\theta_1\theta_2\theta_3(List\ \alpha^{\theta_1}) \times (List^{\theta_2}\ \alpha^{\theta_3}) \rightarrow (List^{\theta_2}\ \alpha^{\theta_1,\theta_3}) \\ \texttt{reverse}: & \forall\alpha.\forall\theta(List\ \alpha^{\theta}) \rightarrow (List\ \alpha^{\theta}) \end{array}$$

That is, the result from append shares elements of both its argument lists, but only the second argument's structure is accessible from the result. For reverse, the result shares the elements of the argument list, but a new structure is created. Further, note that for the following version of reverse

```
reverse(xs) = if (xs=Nil)
              then xs
              else append(reverse(tail(xs)),Cons(head(xs),Nil))
in reverse(as)
```

the inferred type is

$$\texttt{reverse}: \ \forall\alpha.\forall\theta_1\theta_2(List^{\theta_1}\ \alpha^{\theta_2}) \rightarrow (List^{\theta_1}\ \alpha^{\theta_2})$$

i.e., even though the value of `xs` is known to be `Nil` at the `then` branch, this information is not known to the inferencer, which only knows about types, and not values.

4.3 Principal Typing

Lemma 1 All dependencies to a class have been solved before the collection is considered for extended-type inference $\mathcal{W}^l$, $\mathcal{W}^v$.

Proof This is a direct consequence of identifiers being classified into collections, and these collections being sorted topologically with respect to their interdependencies. □

Lemma 2 Suppose $\hat{\chi}_1^v = \mathcal{W}^v\ Te$, and $T \vdash e : \hat{\chi}_2^v$. Then $\hat{\chi}_2^v \succeq \hat{\chi}_1^v$. A similar lemma holds for the relation of $\mathcal{W}^l$ and $\overset{l}{\vdash}$.

Proof In general, $\mathcal{W}^v$ will assume a (precomputed) approximation to the type for every let-bound identifier defined inside e (rather than recursively computing their extended type). Even when the types assumed are the actual types, it can be proven by structural induction that the computed type can be instantiated to any type inferred by the inference rules. □

Lemma 3 Suppose $T \vdash e : \sigma$, σ being standard Hindley-Milner type. Then there is a finite lattice of extended types corresponding to σ, which only use free liveness labels appearing in T, and new bound liveness labels.

Proof By structural induction on the structure of the type of the expression. Type is finite, and hence it cannot contain an infinite number of new labels (there can be at most one new label per type node). □

Lemma 4 Let $\{v_1, \ldots, v_m, f_1, \ldots, f_n\}$ be a collection of let-bound identifiers of the expression e, and further assume that under assumptions T all (non-recursive) dependencies of this collection have been successfully typed. Further, assume that the types $\{\chi_1^v, \ldots \chi_m^v, \chi_1^f, \ldots \chi_n^f\}$ is the conjunctive fixpoint for this set of identifiers. Then these types can be inferred by appropiate application of the inference rules.

Proof This is proved by using induction on the structure of the expression. □

Theorem 1 $\mathcal{T}_p\ e$ computes the principal type for each let-bound identifier in the expression e.

Proof This is a consequence of the previous lemmas. □

5 Conclusion

For first order closed expressions, the extended type system is just a simple extension of the Hindley-Milner type system. Even though the types presented here are non-shallow on liveness labels, there are corresponding shallow versions for first order type. The reason is that the scope of all quantified variables extend to the right of the type (the intersection of the scopes of two variables is always one of them), therefore, a simple 'lifting' of the quantifiers to the outside of the type, with a suitable renaming of labels suffixes to transform them to a shallow type. On the other hand, extended types for the higher-order case cannot be made shallow, and it would be a mistake to try to do so.

On results related to type inference, we have shown that a 'most-general extended-type' exists for the first-order language presented here.

References

[Bak90] Henry G. Baker. Unify and conquer (garbage, updating, aliasing, ...) in functional languages. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programmming.* ACM, 1990.

[BW88] R. Bird and P. Wadler. *Introduction to Functional Programming.* Prentice Hall. 1988.

[DM82] L. Damas and R. Milner. Principle type schemes for functional languages. In *9th ACM Symposium on Principles of Programming Languages.* ACM, August 1982.

[GH90] Juan Carlos Guzmán and Paul Hudak. Single-threaded polymorphic lambda calculus. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*. IEEE, 1990.

[GL86] D.K. Gifford and J.M. Lucassen. Integrating functional and imperative programming. In *Proceedings of 13th ACM Symposium on Principles of Programming Languages*. ACM, 1986.

[Hin78] R. Hindley. The principal type scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1978.

[JG89] Pierre Jouvelot and David K. Gifford. Communication effects for message-based concurrency. Technical Report LCS TM-386, MIT, February 1989.

[KM89] Tsung-Min Kuo and Prateek Mishra. Strictness analysis: A new perspective based on type inference. Technical Report 89/13, The State University of New York at Stony Brook, 1989.

[LG88] J.M. Lucassen and D.K. Gifford. Polymorphic effect systems. In *Proceedings of 15th ACM Symposium on Principles of Programming Languages*. ACM, 1988.

[Mil78] R.A. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.