Participants' Proceedings of the Workshop Types for Program Analysis

Hanne Riis Nielson
Kirsten Lackner Solberg
(Editors)

May 1995

Participants proceedings of the workshop

Types for Program Analysis

As a satellite meeting of the TAPSOFT'95 conference we organized a small workshop on program analysis. The title of the workshop, "Types for Program Analysis", was motivated by the recent trend of letting the presentation and development of program analyses be influenced by annotated type systems, effect systems, and more general logical systems. The contents of the workshop was intended to be somewhat broader; consequently the call for participation listed the following areas of interest:

- specification of specific analyses for programming languages,
- the role of effects, polymorphism, conjunction/disjunction types, dependent types etc. in specification of analyses,
- algorithmic tools and methods for solving general classes of type-based analyses,
- the role of unification, semi-unification etc. in implementations of analyses,
- proof techniques for establishing the safety of analyses,
- relationship to other approaches to program analysis, including abstract interpretation and constraint-based methods,

- exploitation of analysis results in program optimization and implementation

The submissions were not formally refereed; however each submission was read by several members of the program committee and received detailed comments and suggestions for improvement. We expect that several of the papers, in slightly revised forms, will show up at future conferences.

The workshop took place at Aarhus University on May 26 and May 27 and lasted two half days. This volume constitutes the participants proceedings of the meeting. Organization of the conference was made possible due to partial support from the DART project (sponsored by the Danish Science Research Council) and the LOMAPS project (sponsored by ESPRIT Basic Research).

Programme Committee:

Fritz Henglein Computer Science Department University of Copenhagen Denmark	Daniel Le Métayer IRISA / INRIA Rennes France
Flemming Nielson Computer Science Department Aarhus University Denmark	David Wright Department of Computer Science University of Tasmania Australia

Organization Committee:

Hanne Riis Nielson Computer Science Department Aarhus University Denmark	Kirsten Lackner Solberg Computer Science Department Aarhus University Denmark

Contents

A Calculus of Tagged Types, with applications to process languages

Bernard Berthomieu
Camille le Moniés de Sagazan
LAAS / CNRS, 7, Avenue du Colonel Roche, 31077 Toulouse Cedex, France
Tel: (+33) 6133 62 00, Fax: (+33) 6133 64 11, Telex: 52@30 F (UASTSE)
e-mail: Bernard.Berthomieu@laas.fr, Camille.le.Monies.de.Sagazan@laas.fr

Abstract

Tagged Types encode some families of types indexed by labels. Like Wand's row types, they make use of specific variables from which new fields can be extracted, and like Rémy's record types, the types introduced in these fields can be parameterized. However, the logical and technical treatment of tagged types is original; it is claimed that they combine the intuitiveness of row types with most of the expressiveness of record types. Their use is illustrated by the design of polymorphic typing systems for various process calculi extending CCS. Processes are assigned types which associate polymorphic types with all communication labels.

1 Introduction

1.1 Types for tagged expressions

Typing record expressions or some behavior expressions, or inferring signatures in module systems, require collections of types associated with labels. A number of proposals have been done to give such structures some extensions capabilities. Wand introduced *row* variables in [20] to type records with partial knowledge of their fields. Rémy [13] subsequently used infinitary products and proposed a convenient polymorphic type system for record expressions. A connection between the two formalisms is made clear in [21] where row types are presented as a handy notation for some of Rémy's record types.

In row types notation, a function taking a record as argument and extracting the content of its field labelled $\mathtt{a}$ has type $\{\mathtt{a} : \mathrm{P}(\alpha)\}\Delta \to \alpha$ where encapsulation by P means that field $\mathtt{a}$ is required and the row variable Δ implicitly associates some type variable $\alpha_{\mathtt{b}}$ with every label $\mathtt{b} \neq \mathtt{a}$. In Rémy's notation, it has type $\{\mathtt{a} : \mathrm{P}(\alpha); \beta\} \to \alpha$, where β is a "pattern" for the types in other fields than that labelled $\mathtt{a}$.

Record types and row types may be read as total mappings, assigning types to all labels. Consistently with Rémy's treatment, row variables may be read as partial mappings. The row type $\{\mathtt{a} : \mathrm{P}(\mathrm{int})\}\Delta$, for instance, associates a type with every possible record label but, in that context, the row variable Δ does not assign any type to label a since one is already assigned to it. This interpretation implies a well-formedness condition for types excluding types like e.g. $\Delta \to \{\mathtt{a} : \mathrm{int}\}\Delta$.

The Tagged Types we introduce use row variables, but these are interpreted as total mappings; well-formedness of types becomes a simple grammatical constraint. Label replications (as in {p: int} {p: β}Ω, obtained from {p: int}Δ by substituting Δ) are handled by considering that a field supersedes any inner fields bearing the same label. In addition, row variables are associated with type schemes used as patterns for the types implicitly assigned by the variables; this gives tagged types most of the flexibility of Rémy's record types. From an axiomatization of their equality, we derive unification algorithms that extend in a simple way the standard techniques.

1.2 Typing process calculi

Tagged types were developed to typecheck parallel programs written in LCS [3], a language combining ML and an higher order version of the Calculus of Communicating System [9] with behavior passing and parametric channels.

There are basically two classes of process programming languages, according to whether or not communication channels are considered first-order values.

Languages in the former class include the π-calculus [10], CML [15], FACILE [5] and Plain CHOCS [17]. In these languages, channels are typically typed like locations in a store; processes are merely checked in an environment assigning types to channels and other variables, and are given a trivial *process* type (see e.g. [7]). Alternatively, [12] annotate functional types with behavioral informations from which various static analyses, including typing, can be performed.

Calculi and languages in the latter class include CHOCS [16], TPL [11] and LCS [3]. The "channels as locations" analogy would give here a poor typing system, rejecting many useful programs. A better approach is to see processes as records of channels and to assign each process the set of types expected on the communication labelsit uses. This is basically the approach of [11].

Computing statically that set of labels is not always possible, in particular in languages allowing process-passing or permitting to define process combinators. Such features can be handled by making the type of a process assign a type information to all possible communication labels, and not only to those used by the process; such process types are conveniently encoded by tagged types. This is the approach taken for LCS; it bears strong similarities with the way record expressions are typed in [21] or [13].

1.3 Plan of the paper

Tagged Types are introduced in Section 2, including unification techniques. Section 3 illustrates their use for typing CCS programs and various higher order extensions. We conclude with some comparisons with related work. Most of the results are given here without proofs, these will be included in a longer version of this paper. More details can be found in [2], an earlier attempt at formalizing tagged types, or in the second author's forthcoming thesis [6].

2 Tagged Types

2.1 The language of tagged types

Given a countable set of labels $\Sigma = \{\mathrm{p}, \mathrm{q}, \ldots\}$, *types* τ, of which the *tagged types* ρ are a subset and *type schemes* σ obey the following grammar:

$$\begin{array}{lcll}
\tau & ::= & \alpha, \beta, \gamma, \ldots & \text{plain type variables} \\
 & | & \mathsf{a}, \mathsf{b}, \mathsf{c}, \ldots & \text{plain constant types} \\
 & | & \tau \rightarrow \tau & \text{function type} \\
 & | & \rho & \text{tagged types} \\
 & & & \\
\rho & ::= & \Delta^{\sigma}, \Theta^{\sigma'}, \ldots & \text{tagged type variables, } \sigma, \sigma' \text{ closed} \\
 & | & \nabla^{\mu}, \nabla^{\mu'}, \ldots & \text{tagged type constants, } \mu, \mu' \text{ monotypes} \\
 & | & \{p : \tau\}\rho & \text{field prefixing, for each } p \in \Sigma \\
\sigma & ::= & \tau \mid \forall \alpha^{s}.\sigma & \text{type schemes}
\end{array}$$

The types are classified into sorts; infinitely many variables are available at each sort. Plain variables and plain constants have the *plain* sort $\mathcal{U}$. A tagged variable Δ^{σ} or tagged constant ∇^{σ} has the *tagged* sort $\mathcal{T}(\sigma)$; each closed type scheme defines a tagged sort. Sort assignment for the other types will soon be made precise. A *monotype* is a type in which no variable occurs (of any sort). Though this is not essential, it is assumed that tagged variables have polymorphic sorts (i.e. the closed type scheme σ in Δ^{σ} contains at least one variable).

Tagged types have a number of *fields* prefixing a tagged variable or tagged constant (called the *extension*); each field associates a type with a label. The type $\{\mathrm{p}_1 : \tau_1\} \ldots \{\mathrm{p}_k : \tau_k\}\rho$ is also written $\{\mathrm{p}_1 : \tau_1 \ldots \mathrm{p}_k : \tau_k\}\rho$ when type ρ has no fields and labels p_1 are pairwise distinct. Sort annotations are omitted for tagged variables having sort $\mathcal{T}(\forall\alpha.\alpha)$: Δ, Θ, stand for the variables $\Delta^{\forall\alpha.\alpha}$, $\Theta^{\forall\alpha.\alpha}$.

As usual, type schemes are types possiblyquantified at the outermost. $\bar{\tau}$ denotes the closure of type τ, obtained by quantifying all of its variables.

Finally, we restrict here the possible tagged sorts to those not including themselves any tagged variables. This is for the sake of simplicity; the treatment developed in the following sections is easily extended to higher order tagged sorts.

2.2 Substitutions

$[\,]$ is the identity substitution; $[\tau_1/\alpha_1^s, \ldots, \tau_k/\alpha_k^{s'}]$ replaces variables $\alpha_1^s, \ldots, \alpha_k^{s'}$ by types $\tau_1, \ldots, \tau_k$, respectively; $S_2 \mathrm{o} S_1$ is the composition of S_1 and S_2. $\leq$ is the substitution instance preorder ($\tau \leq \tau' \Leftrightarrow \exists S.\ \tau = S\ \tau'$), and $\equiv$ is the associated equivalence relation ($\tau \equiv \tau' \Leftrightarrow \tau \leq \tau' \wedge \tau' \leq \tau$).

To define substitutions, it is convenient to cast our types into the framework of order-sorted languages: Terms in an order sorted language are classified into sorts, and the set of sorts is equipped with a partial ordering $\subseteq$ (often called sort inclusion). Substitutions are restricted to those preserving sorts: For any substitution S and variable v, we must have $\mathrm{Sort}(S\ \mathrm{v}) \subseteq \mathrm{Sort}(\mathrm{v})$.

The sort membership rules for variables and constants were explained in Section 2.1. Function types have the plain sort $\mathcal{U}$. A tagged type $\{\mathrm{p} : \tau\}\rho$ has sort $\mathcal{T}(\sigma')$ when $\mathcal{T}(\overline{\tau}) \subseteq \mathcal{T}(\sigma')$ and $\mathcal{T}(\sigma) \subseteq \mathcal{T}(\sigma')$, where $\mathcal{T}(\sigma)$ is the sort of type ρ. Relation $\subseteq$ includes all pairs $\mathcal{T}(\sigma) \subseteq \mathcal{U}$, for any σ, and all pairs $\mathcal{T}(\overline{\tau}) \subseteq \mathcal{T}(\overline{\tau'})$ with $\tau \leq \tau'$.

In other words, a plain variable may be substituted by any type while a tagged variable of sort $\mathcal{T}(\overline{\tau})$ may only be substituted by tagged types in which the type in each field and the exponent of the extension are instances of τ. E.g. let $\tau = \{\mathrm{p} : \alpha\}\Delta^{\forall\delta\beta.\delta\rightarrow\beta}$; then the types $\{\mathrm{p} : \alpha\}\ \{\mathrm{q} : \lambda \rightarrow \mu\}\Theta^{\forall\delta\beta.\delta\rightarrow\beta}$ and $\{\mathrm{p} : \mathsf{e}\}\nabla^{\mathsf{b}\rightarrow\mathsf{c}}$ are substitution instances of τ, but neither $\{\mathrm{p} : \alpha\}\Delta^{\forall\beta.\beta}$ nor $\{\mathrm{p} : \alpha\}(\gamma \rightarrow \delta)$ are.

A *renaming* is a substitution mapping variables to variables and injective on its domain; renamings are written R, R', etc. We have $\tau \equiv \tau'$ iff $\exists\ R.\ \tau = R\ \tau'$. A substitution is *nonexpansive* when it maps tagged variables to tagged variables; nonexpansive substitutions are written N, N', etc.

2.3 Equality

Models of tagged types

To avoid a semantics of tagged types tied to a specific programming notation, we define an extensional equality that we assume valid in all models of closed type schemes. Equality of closed type schemes ($=_g$) is defined from their monotype instances (Π, Π' are tagged monotypes):

$$\sigma =_g \sigma' \Leftrightarrow (\forall \mu \leq \sigma \Rightarrow \exists \mu' \leq \sigma'.\ \mu =_g \mu') \wedge (\forall \mu' \leq \sigma' \Rightarrow \exists \mu \leq \sigma.\ \mu =_g \mu')$$

$$\begin{array}{lll}
\mathsf{a} =_g \mathsf{a} & \text{(for each plain constant } \mathsf{a}) & \\
\mu_1 \to \mu'_1 =_g \mu_2 \to \mu'_2 & \text{if} & \mu_1 =_g \mu_2 \wedge \mu'_1 =_g \mu'_2 \\
\Pi =_g \Pi' & \text{if} & \forall \mathrm{p} \in \Sigma.\ \Pi(\mathrm{p}) =_g \Pi'(\mathrm{p}) \\
\quad \text{with } \Pi(p) \text{ obtained by:} & (\{\mathrm{p} : \mu\}\Pi)(\mathrm{p}) = \mu & \\
 & (\{\mathrm{q} : \mu\}\Pi)(\mathrm{p}) = \Pi(\mathrm{p}) \qquad (\mathrm{q} \neq \mathrm{p}) & \\
 & \nabla^{\mu}(\mathrm{p}) = \mu &
\end{array}$$

Monotype tagged types are seen as mappings; ∇^{μ} is a constant mapping; $\{\mathrm{p} : \mu\}\Pi$ stands for the mapping obtained from Π by replacing its $(\mathrm{p}, \_)$ pair by (p, μ). We axiomatize in the sequel equality $=_g$.

$\sim$-equality

Relation $\sim$ is the smallest congruence including identity and the pairs from:

(permutation)	$\{\mathrm{p} : \tau\}\{\mathrm{q} : \tau'\}\rho \sim \{\mathrm{q} : \tau'\}\{\mathrm{p} : \tau\}\rho$	(for any p,q: $\mathrm{p} \neq \mathrm{q}$)
(pruning)	$\{\mathrm{p} : \tau\}\{\mathrm{p} : \tau'\}\rho \sim \{\mathrm{p} : \tau'\}\rho$	(for any p)
(μ-expansions)	$\nabla^{\mu} \sim \{\mathrm{p} : \mu\}\nabla^{\mu}$	(for any p)

Fields may appear in any order. A field may be removed when it follows another field bearing the same label; such fields are said *hidden.* The last axiom expresses how to extract a field out of a tagged constant.

$\sim$-equality is preserved by substitutions. $|\tau|$ is the *normal form* of type τ, obtained from τ by removing its hidden fields. $\mathcal{V}(\tau)$ is the set of variables occurring in $|\tau|$.

Expansion equivalence

We would like to express some extensionality axiom for tagged variables, similar to the one above for tagged constants. For this, we need some method to control introduction of variables:

An *expansion component* is a substitution $[\{\mathrm{p} : \tau'\}\Delta^{\bar{\tau}}/\Delta^{\bar{\tau}}]$, with $\tau' \equiv \tau$ and $(\mathcal{V}(\tau) \cup \{\Delta^{\bar{\tau}}\}) \cup \mathcal{V}(\tau') = \emptyset$. Given a set W of *protected variables*, an *expansion basis away from* W is a set of substitutions such that (1): It holds the empty substitution and exactly one component $[\{\mathrm{p} : \tau'\}\Delta^{\bar{\tau}}/\Delta^{\bar{\tau}}]$ for each pair $(\Delta^{\bar{\tau}}, p)$, (2): For any two components $[\{\mathrm{p} : \tau\}\Delta^{\sigma}/\Delta^{\sigma}]$ and $[\{\mathrm{q} : \tau'\}\Theta^{\sigma'}/\Theta^{\sigma'}]$ in the set, we have $\mathcal{V}(\tau) \cap \mathcal{V}(\tau') = \emptyset$ and $(\mathcal{V}(\tau) \cup \mathcal{V}(\tau')) \cap W = \emptyset$ and (3): it is closed by composition of substitutions.

For each W, Ξ^{W} denotes the union of all expansion basis away from W; the elements of Ξ^{W} are called *expansions away from* W.

From this we define the *expansion preorder* $\lesssim_\eta$, and *expansion equivalence* $\sim_\eta$, both away from W, by:

$$\begin{aligned} \tau \lesssim_\eta \tau' \quad &\Leftrightarrow \quad \exists \mathrm{W}, X \in \Xi^{\mathrm{W}}.\ \mathcal{V}(\tau') \subset \mathrm{W} \wedge \tau \sim X\tau' \\ \tau \sim_\eta \tau' \quad &\Leftrightarrow \quad \exists \tau''\ .\ \tau \lesssim_\eta \tau'' \wedge \tau' \lesssim_\eta \tau'' \end{aligned}$$

$\sim_\eta$ is an equivalence relation. $\tau \lesssim_\eta \tau'$ means that τ may be obtained from τ' by making some of its fields explicit. The types introduced follow from the sort of the variable expanded; the definition of Ξ^{W} prevents expansions to introduce a variable in fields with different labels, or variables already used in the type expanded.

Instances and $\cong$-equivalence

The substitution instance preorder $\leq$ is too restrictive for our purpose; we want in addition to consider types modulo $\sim$ and expansion-equivalence. We define thus a coarser preorder $\lesssim$ (read *less* or *equally general*) as the least preorder obeying:

$$
\begin{array}{l}
\tau \leq \tau' \;\Leftrightarrow\; \tau \lesssim \tau' \\
\tau \sim \tau' \;\Leftrightarrow\; \tau \lesssim \tau' \\
(\exists\ \tau_1, \tau_2 . \tau \gtrsim_\eta \tau_1 \lesssim \tau_2 \lesssim_\eta \tau') \Rightarrow \tau \lesssim \tau'
\end{array}
$$

$\cong$ is the associated equivalence relation (read *equally general*) by:

$$\tau \cong \tau' \;\Leftrightarrow\; \tau \lesssim \tau' \wedge \tau' \lesssim \tau$$

Example 1: Consider the following types (all tagged variables have sort):

(1)	$\{\mathrm{p}:\alpha\}\{s:\gamma\}\Delta \rightarrow \{\mathrm{p}:\beta\}\{s:\gamma\}\Delta$	(2)	$\{\mathrm{p}:\alpha\}\Delta \rightarrow \{\mathrm{p}:\beta\}\Delta$
(3)	$\{\mathrm{p}:\alpha\}\Delta \rightarrow \Delta$	(4)	$\{\mathrm{p}:\beta\}\{s:\lambda\}\Theta \rightarrow \{\mathrm{p}:\omega\}\{s:\lambda\}\Theta$
(5)	$\{\mathrm{p}:\beta\}\Theta \rightarrow \{\mathrm{p}:\omega\}\Theta$	(6)	$\Theta \rightarrow \{\mathrm{p}:\omega\}\Theta$

We clearly have (1) $\lesssim_\eta$ (2) $\lesssim_\eta$ (3) and (1) $\sim_\eta$ (2) $\sim_\eta$ (3), as well as (4) $\sim_\eta$ (5) $\sim_\eta$ (6), but neither (3) $\lesssim_\eta$ (6), nor (6) $\lesssim_\eta$ (3). However, we have (3) $\lesssim$ (6) and (6) $\lesssim$ (3); all types are equivalent by $\cong$. None of them are related by $\sim$.

$\lesssim$ and $\cong$ are characterized as follows in terms of substitutions (R is a renaming and N is a *nonexpansive* substitution):

$$
\begin{array}{lcl}
\tau \lesssim \tau' & \Leftrightarrow & \exists \mathrm{W}, X \in \Xi^{\mathrm{W}}, N.\ \mathcal{V}(\tau) \cup \mathcal{V}(\tau') \subset \mathrm{W} \wedge X\tau \sim (NoX)\tau' \\
\tau \cong \tau' & \Leftrightarrow & \exists \mathrm{W}, X \in \Xi^{\mathrm{W}}, R.\ \mathcal{V}(\tau) \cup \mathcal{V}(\tau') \subset \mathrm{W} \wedge (RoX)\tau \sim X\tau'
\end{array}
$$

Equality of type schemes and canonical forms

Equality of closed type schemes is defined by $\bar{\tau} \cong \bar{\tau'} \Leftrightarrow \tau \cong \tau'$. That equality is sound versus the extensional equality $=_g$ discussed earlier: $\sigma \cong \sigma'$ implies $\sigma =_g \sigma'$. It is conjectured that the converse implication is also true.

Several canonical forms can be defined for closed type schemes. They all follow from the fact that the number of types in normal form which are related by $\cong$ to some type, and are maximal by preorder $\lesssim_\eta$, is finite (modulo a renaming of variables and a permutation of fields). For the types in Example 1, for instance, there are exactly two such maximal classes, including types (3) and (6), respectively. Any total ordering on maximal classes defines a canonical form; the greatest lower bound of these maximal classes by $\lesssim_\eta$, defines an other canonical form. In Example 1, that latter class would hold types (2) and (5).

2.4 Unification

$\sim$**-unification**: A substitution U is a $\sim$-*unifier* of (τ_1, τ_2) iff we have $U\tau_1 \sim U\tau_2$. In addition, U is a $\sim$-*most general unifier* ($\sim$-*mgu* for short) iff for any other $\sim$- unifier V of (τ_1, τ_2), we have $V\tau_1 \lesssim U\tau_1$.

If U and U' are two $\sim$-mgus of (τ_1, τ_2), then we must have $U\tau_1 \cong U'\tau_1$.

Theorem: There is an algorithm that returns for any types τ and τ' a $\sim$-principal $\sim$-unifier, or indicates failure if these types are not $\sim$-unifiable.

We give a proof sketch by reducing $\sim$-unification to an order-sorted vest of the standard first order unification problem:

In what follows, $S\tau_i \sim S'\tau_i$ stands for $S\tau_1 \sim S'\tau_1 \wedge S\tau_2 \sim S'\tau_2$ and all expansions are assumed away from the variables occurring in the types to which they are applied, as well as away from all variables in the surrounding context when unification is used as part of a more general process.

From the characterization of preorder $\lesssim$, finding a $\sim$-mgu of (τ_1, τ_2) is equivalent to finding a nonexpansive substitution N, and an expansion X such that:

(i) $\quad (N\text{o}X)\tau_1 \sim (N\text{o}X)\tau_2$

(ii) $\quad \forall N', X'.(N'\text{o}X')\tau_1 \sim (N'\text{o}X')\tau_2$
$\qquad \Rightarrow \exists X'', W.(X''\text{o}N'\text{o}X')\tau_i \sim (W\text{o}N\text{o}X)\tau_i$

If such an N and X are found, then $(N\text{o}X)$ is a $\sim$mgu of (τ_1, τ_2). That problem can be shown equivalent to finding N and X such that:

(i') $\quad (N\text{o}X)\tau_1 \sim (N\text{o}X)\tau_2$

(ii') $\quad \forall N'.(N'\text{o}X)\tau_1 \sim (N'\text{o}X)\tau_2 \Rightarrow \exists W.(N'\text{o}X')\tau_i \sim (W\text{o}N\text{o}X)\tau_i$

(iii) $\quad \forall N', X'.N'$ $\sim$-mgu of $(X'\text{o}X\tau_1, X'\text{o}X\tau_2)$
$\qquad \Rightarrow \exists X''.(N'\text{o}X'\text{o}X)\tau_i \sim (X''\text{o}N\text{o}X)\tau_i$

A sufficient condition to guarantee the commutation property (iii) is that expansion X makes all tagged variables in (τ_1, τ_2) bear fields for exactly the same set of labels. Further, given such an expansion X, $(X\tau_1, X\tau_2)$ is necessarily equal by $\sim$ to some pair (τ_1', τ_2') holding no hidden fields, in which tagged variables and constants are all preceded by fields for the same set of labels, and these fields occur in the same label-order. Since $\sim$ is preserved by substitutions and $\tau_i' \sim X\tau_i$, an equivalent formulation of the previous problem is to find a nonexpansive substitution N such that:

(i') $\quad N\tau_1' = N\tau_2'$

(ii") $\quad \forall N'.\ N'\tau_1' = N'\tau_2' \Rightarrow \exists W. N'\tau_i' = (W \mathrm{o} N)\tau_i'$

That is of finding a nonexpansive and sort preserving unifier of τ_1' and τ_2' which is most-general in the usual sense. The order-sorted discipline of Section 2.2 is easily embedded in the standard unification algorithms, without affecting their correctness. Plain variables may be substituted by any type. Unifying two tagged variables, say $\Delta^{\overline{\tau}}$ and $\Theta^{\overline{\nu}}$, consists of substituting a third variable $\Theta^{\overline{\omega}}$ for both (asserted not to occur yet in the unification context nor in any expansion), with ω determined as the most general type which is an instance of both σ and δ; type ω is naturally obtained from the unification of σ and δ. If ω is reduced to a monotype, we should substitute both $\Delta^{\overline{\tau}}$ and $\Theta^{\overline{\nu}}$ by the tagged constant ∇^{ω} instead. Unifying a tagged variable $\Delta^{\overline{\tau}}$ with a tagged constant ∇^{μ} is substituting it by the constant, if μ can be unified with τ. ■

This rudimentary algorithm suffices to prove existence of $\sim$-unification algorithms. Another algorithm is suggested in Appendix, convenient for practical purposes. It integrates the expansion and normalization steps and relies on a weaker sufficient condition than that used above to fulfill condition (iii).

As an example, applied to the pair:

$$\{\mathrm{p} : \Psi\}\{r : \mathsf{b} \to \alpha\}\Delta^{\forall\alpha.\mathsf{a}\to\alpha}, \{\mathrm{q} : \delta\}\{r : \beta\}\Theta^{\forall\beta\gamma\nu.\beta\to\gamma\to\nu})$$

Expansion and normalization produces the pair:

$$\begin{array}{l}(\{\mathrm{p} : \Psi\}\{q : \mathsf{a} \to \alpha_q\}\{r : \mathsf{b} \to \alpha\}\Delta^{\forall\alpha.\mathsf{a}\to\alpha}, \\ \{\mathrm{p} : \beta_{\mathrm{p}} \to \gamma_{\mathrm{p}} \to \nu_{\mathrm{p}}\}\{\mathrm{q} : \delta\}\{r : \beta\}\Theta^{\forall\beta\gamma\nu.\beta\to\gamma\to\nu})\end{array}$$

And unification returns (after normalization):

$$\{\mathrm{p} : \beta_{\mathrm{p}} \to \gamma_{\mathrm{p}} \to \nu_{\mathrm{p}}\}\{q : \mathsf{a} \to \alpha_{\mathrm{q}}\}\{\mathrm{r} : \mathsf{b} \to \alpha\}\Omega^{\forall\gamma\nu.\mathsf{a}\to\gamma\to\nu}$$

The variables introduced by expansions are arbitrary, but must be different from those in use in the types and those introduced by unification of tagged variables.

Similarly, from the pair: $(\Delta \to \Theta \to \{\mathrm{p} : \mathsf{a}\}\Delta, \Delta \to \Theta \to \{\mathrm{p} : \mathsf{a}\}\Theta)$

Unification produces: $\{\mathrm{p} : \delta\}\Omega \to \{\mathrm{p} : \theta\}\Omega \to \{\mathrm{p} : \mathsf{a}\}\Omega$

Note in this last example that the naive $\sim$-unifier $[\Omega/\Delta, \Omega/\Theta]$ is not a $\sim$-mgu. It leads to type $\Omega \to \Omega \to \{\mathrm{p} : \mathsf{a}\}\Omega$, expansion-equivalent to $\{\mathrm{p} : \delta\}\Omega \to \{\mathrm{p} : \delta\}\Omega \to \{\mathrm{p} : \mathsf{a}\}\Omega$ which is clearly less general by $\lesssim$ than the result of unification.

2.5 Further issues and Related work

Tagged types were compared to row types in the introductory section. In the general allowing higher-order sorts (not discussed here), tagged Types and the generalized record types in [14] have different expressiveness since record types do not admit higher order expansion patterns while tagged types do not allow type variables to be shared between expansion patterns as in the record type $\{\alpha\} \to \{\beta\} \to \{\alpha \to \beta\}$. However, when neither of these features are used, one can easily pass from one of the formalisms to the other.

The usefulness of higher order sorts is questionable as long as no applications taking advantage of them are found. But nontrivial sorts are undoubtedly useful. They serve to encode inheritance in type systems for records, and some of the type systems discussed in Section 3 take advantage of them as well.

As a summary, well-formedness of tagged types reduces to a simple grammatical constraint, tagged variables have a meaning independent of any context, extensibility is simply explained, tagged types can be unified with simple extensions of the standard algorithms, and closed type schemes admit a variety of canonical forms. We believe that tagged types provide a more intuitive and technically simpler alternative to the existing treatments.

3 Typing CCS and Higher Order variants

3.1 The Calculus of Communicating Systems

Given a language of message expressions e and a countable set of labels Σ, our starting process calculus is CCS [9], extended by local declarations (x is a message variable, X is a process variable, a and b are labels in Σ):

$$\begin{aligned} P ::= {} & a(x).P \mid \overline{b}e.P \mid \tau.P \mid \Sigma_{i\in I}P_i \mid P \mid P' \mid P\backslash a \mid P[b/a] \mid \textbf{if } e \textbf{ then } P \textbf{ else } P' \\ & \mid \textbf{letrec } X = P \textbf{ in } Q \mid X \mid \textbf{let } X = P \textbf{ in } Q \end{aligned}$$

Message expressions e should include two constants encoding a boolean type. CCS expressions denote processes (or behaviors). They are built, possibly recursively, from the empty summation (written **0**), action prefixing, restriction, renaming and compositions. Actions include proposing a message on a label ($\overline{a}$e.P), accepting a message on a label (a(x).P), and performing an internal move (τP).

Processes communicate by rendez-vous. Restrictions and relabelling delimit the scope of label instances. $P\backslash a$ delimits the scope of the instances of a in P to P. P [a/b] makes actions using labels a and b in P appear to the enclosing context as all using label a (but not within P).

A simple rule ensures type-safe communications between processes: Whenever their scopes intersect, two label instances must transmit messages of the same type. The rule is sufficient though not necessary; it is only necessary for the pairs which may actually be involved in a communication, but this cannot be generally inferred at compile-time (unless severely restricting message expressions).

3.2 Typing CCS expressions

Behavior expressions will be assigned *behavior types*, which associate a type with every possible communication label. Behavior types are conveniently defined as a particular class of tagged types. Two sorts are required: a sort $\mathcal{M}$ of *message* types, and a *process* sort $\mathcal{P}$ that can be defined as the tagged

$\frac{-}{A\vert-x{:}\mu}A(x)\geq\mu$	$\frac{-}{A\vert-X{:}\pi}A(X)\geq\pi$	$\frac{A\vert-P{:}\pi}{A\vert-P{:}\pi'}\ \pi\sim\pi'$ *(equal)*
$\frac{A(X)\geq\pi}{A\vert-X{:}\pi}$	$\frac{A\oplus\langle x{:}\mu\rangle\vert\text{-}P : \{\mathrm{a} : \mu\}\pi}{A\vert-a(x).P{:}\{\mathrm{a}{:}\mu\}\pi}$	$\frac{A\vert-e{:}\mu\ \ A\vert-P{:}\{\mathrm{a}{:}\mu\}\pi}{A\vert-\bar{a}e.P{:}\{\mathrm{a}{:}\mu\}\pi}$
$\frac{A\vert-P{:}\{\mathrm{a}{:}\mu\}\pi}{A\vert-P\backslash\mathrm{a}{:}\{\mathrm{a}{:}\mu\}\pi}$	$\frac{A\vert-P{:}\{\mathrm{a}{:}\mu\}\{\mathrm{b}{:}\mu\}\pi}{A\vert-P[\mathrm{a/b}]{:}\{\mathrm{a}{:}\mu\}\{\mathrm{b}{:}\mu'\}\pi}$	$\frac{A\vert-\mathrm{b{:}bool}\ \ A\vert-P{:}\pi A\vert-P'{:}\pi}{A\vert-\ \textbf{if}\ b\ \textbf{then}\ P\ \textbf{else}\ P'{:}\pi'}$
$\frac{(\forall i\in I)A\vert-P_i{:}\pi}{A\vert-\Sigma_{i\in I}P_i{:}\pi}$	$\frac{A\vert-P{:}\pi\quad A\oplus\langle X{:}Gen(A,\pi)\rangle\vert-Q{:}\pi'}{A\vert-\mathrm{let}\ \ X{=}P\ \ \mathrm{in}Q{:}\pi'}$	
$\frac{A\vert-P{:}\pi\ \ A\vert-P'{:}\pi}{A\vert-P\vert P'{:}\pi}$	$\frac{A\oplus\langle X{:}\pi\rangle\vert-P{:}\pi\quad A\oplus\langle X{:}Gen(A,\pi)\rangle\vert-Q{:}\pi'}{A\vert-\mathrm{letrec}\ \ X{=}P\ \ \mathrm{in}\ \ Q{:}\pi'}$	

Table 1: Type inference rules for CCS + let

sort $\mathcal{T}(\forall\alpha^{\mathcal{M}}.\alpha^{\mathcal{M}})$. Since CCS does not allow process passing (classes e and P above are disjoint), sorts $\mathcal{M}$ and $\mathcal{P}$ are order-unrelated; process types should not be substituted for message type variables, nor message types for process type variables.

CCS behavior types obey the following grammar in which μ (resp. π) rages over all types of sort $\mathcal{M}$(resp. $\mathcal{P}$):

$$\begin{array}{lcll}
\mu & ::= & \alpha, \beta, \gamma, \ldots & \text{message type variables} \\
 & \vert & \text{bool}, \ldots & \text{some type constants} \\
\pi & ::= & \Delta, \Theta, \ldots & \text{process type variables} \\
 & \vert & \{\text{a:}\mu\}\pi & \text{field prefixing}
\end{array}$$

The most general type scheme for a process is $\forall\Delta.\Delta$. The scheme type $\forall-.\{\mathrm{a} : \mu\}\pi$ (assumed closed) is the type of processes that may communicate massages of type μ on label a. For other labels than a, the constraints are given by type π.

The type inference rules for CCS expressions are shown in Table 3.1. A maps message and process identifiers to types. As in [7], $Gen(A, \pi)$ is the type scheme obtained by universally quantifying the variables that are free in π but not in A. Input bound variables here are typed like lambda-bound variables in [4] or [7].

The rule for relabelling says that labels a and b must have same types before relabelling, and that b has any type after relabelling. The processes involved in compositions must all have the same (process) type. The empty summation has any process type; it is assumed that the initial typing environment assigns the type scheme $\forall\Delta.\ \Delta$ to $\mathbf{0}$. Recursion is typed as in [4], which explains e.g. that the expression (**letrec** A = (p!1.A)\p **in** A) cannot be assigned the scheme $\forall\Delta.\Delta$, but only the weaker $\forall\Delta.\{\mathrm{p} : \mathrm{int}\}\Delta$, though that process cannot communicate through p). The *equal* rule allows to permute fields or remove hidden fields in the process of inference, but expansions are assumed performed by the generic instance rule.

Soundness of the inference system is proven along the lines of [18] by first defining the computations that yield type errors and then proving that if some type could be inferred for some process, then it cannot yield an error when run (the complete proof will appear in [6]). As in [4, 8], type synthesis is reduced to unification. The essential part of the type assignment algorithm is shown in Table 3.2. A maps identifiers to types; Id is the identity substitution. The type assignment algorithm does not rely on generic instantiation to perform the necessary expansions; these are performed by the unification algorithm for tagged types instead (function *Unify*). Function *newtype(s)* returns a new type variable of sort s.

3.3 Behavior Passing

CHOCS [16] or LCS [3] allow process passing, while retaining the semantics of the restriction operator of CCS. To handle this, the only change required to our previous language is to merge the syntactic classes P and e. In addition, we must allow message type variables to be substituted by process types (i.e. the process sort $\mathcal{P}$ becomes included in the message sort $\mathcal{M}$). The type inference rules are as in Table 3.1, with variables X and x assumed in the same syntactic class. These rules enforces the fact that, whenever an expression is used as a behavior expression, it must have a behavior type. The algorithm in Table 3.2 is easily updated to enforce these constraints

```
$W(A,P)$ = case $P$ of
$X$        =>  | if $X \in Dom(A)$ then(Id,$A(X)$)elseFAIL
$\tau.P$   =>  let  $(S_1,\pi)$  $=W(A,P)$
                    $S_2$        $= Unify(\pi,newtype(\mathcal{P})$
               in   $(S_2,S_1,S_2\pi)$
$\overline{a}e.P$ => let $(S_1,\mu)$ $=W(A,e)$
                    $(S_2,\pi)$  $=W(S_1\ A,P)$
                    $S_3$        $Unify\ (\pi,\{$a: $S_2\mu\}\ (newtype(\mathcal{P})))$
               in   $(S_3S_2S_1,S_3\pi)$
$a(x).P$   =>  let  $\mu$        $=newtype(\mathcal{M})$
                    $(S_1,\pi)$  =W(A$\oplus\langle x{:}\mu\rangle$,P)
                    $S_2$        $= Unify(\pi,\{$a: $S_1\mu\}(newtype(\mathcal{P})))$
               in   $(S_2S_1S_2\pi)$
$P\backslash a$ => let $(S_1,\pi)$ $=W(A,P)$
                    $S_2$        $= Unify(newtype(\mathcal{P}),\pi)$
               **in** $(S_2S_1\{$a:$newtype()\}(S_2\pi))$
$P[a/b]$   =>  **let** $(S_1,\pi)$ $=W(A,P)$
                    $\mu$        $= newtype(\mathcal{M}))$
                    $S_2$        $Unify(\pi,\{$a: $\mu\}\{b{:}\mu\}(newtype(\mathcal{P})))$
               in   $(S_2S_1\{b{:}newtype()\}(S_2\pi))$          etc.
```

Table 2: Type assignment algorithm for CCS (fragments)

(expressions like e.g. p!2.4 would be rejected).

Processes like e.g. (a(x).x) cannot be assigned types in that system (the solution τ should solve $\tau = \{\mathrm{a} : \tau\}\Delta$). This problem is equivalent to that of typing (**letrec** f $= \lambda$x.f **in** f) in [4], or that of typing channels communicating themselves in [19]. Clearly, the problem cannot be solved without some form of recursive types, either part of the language of types (we did not investigate the issue for tagged types), or provided at the level of declaration of new type operators (as in ML).

3.4 Parametric Channels

Following a proposal of [1], LCS provides parameterized channels to implement a weak form of delegation. LCS communication ports are constituted of a label, as in CCS, and of an extension (a value admitting equality). Extensions may not appear in restrictions or relabellings. The typing problem with these parametericed channels is essentially the same than for CCS, except that when the scope of two ports intersect, they must have the same extension types too.

To handle these ports, the behavior types of CCS must be enriched with a sort $\mathcal{Q}$ of extension types (possibly a subsort of $\mathcal{M}$). Process types assign to each label a pair of types $\theta \bullet \mu$, with θ of sort $\mathcal{Q}$ and μ of sort $\mathcal{M}$ the process sort $\mathcal{P}$ is redefined as $\mathcal{T}(\forall\alpha^{\mathcal{Q}}\beta^{\mathcal{M}}.\alpha^{\mathcal{Q}} \bullet \beta^{\mathcal{M}})$. Except for this, the treatment is similar to that for CCS.

3.5 Lambda expressions

Adding abstraction and application to our language would permit to define new process combinators. A pipe combinator, connecting a process sending messages on port `out` to another receiving messages on port `inp` could be defined by:

`pipe` $= \lambda$a . λb. (a \tmp [tmp/out] | b \tmp [tmp/imp]) \tmp

These constructions would be typed as in [7], and the necessary inference rules just added to those in Table 3.1. Extending this way the previous process passing variant of CCS would just require to add function types to the message sort $\mathcal{M}$ The principal type scheme inferred for function `pipe` would be:

$$\forall \alpha\beta\gamma\Delta.\{\text{out} : \alpha\}\Delta \to \{\text{inp} :\alpha, \text{tmp} :\beta\}\Delta \to \{\text{tmp} :\gamma\}\Delta$$

4 Conclusion

Tagged types clearly build upon the work of Wand on row types and that of Rémy on record types. The connections with these formalisms were explained in Section 2 and the Introduction. For practical purposes, tagged types combine, we believe, the intuitiveness of row types with the expressiveness of record types. Further, they require a lighter theoretical treatment.

All these formalisms widely extend the class of expressions of programming languages for which types can be mechanically inferred. For process calculi, our results strictly strengthen the "sorting" methods discussed in the literature and permit type inference for parallel programs based on the CCS paradigm. Compared to the types used in [11], ours do not attempt to represent "polarity" of labels (i.e. to distinguish between labels used for input, output, or both), but it fully handles "label" polymorphism. Compared to the method of [19] (for different calculi), ours has the advantage of assigning nontrivial types to process variables, which permit type reconstruction for processes from the types of their constituent processes. This can be considered an advantage in implementations of declarative languages based upon these paradigms.

Tagged types, and the typing methods presented, have been experimented for several years. The typechecker of the implementations of the language LCS is directly based on the unification algorithm found in Appendix, and on the type inference systems and type assignment algorithms discussed in Section 3.

References

[1] E. Astesiano, E. Yucca, “Parametric channels via label expressions in CCS*”, Theoretical Computer Science, Vol 33, 1984.

[2] B. Berthomieu, “Tagged types - a theory of order sorted types for tagged expressions”, Technical Report 93083, LAAS-CNRS, March 1993.

[3] B. Berthomieu, T. Le Sergent, “Programming with behaviors in an ML framework, the syntax and semantics of LCS”, In Programming Languages and Systems - ES-OP’94, Edinburgh, April 1994, LNCS Vol. 788, Springer Verlag, 1994.

[4] L. Damas and R. Milner, “Principal type-schemes for functional programs”, ACM Symposium on Principles of Programming Languages, 1982.

[5] A. Giacalone, P. Mishra, S. Prasad, “Facile: A symmetric integration of concurrent and functional programming”. Int. Journal of Parallel Programming, 18(2), April 1989.

[6] C. le Moniés de Sagazan, Typage Polymorphe des Calculs de Processus á Liaisons Noms-Canaux Dynamiques, Ph. D. Thesis, Univ. of Toulouse, to appear.

[7] X. Leroy, Typage Polymorphe d’un Langage Algorithmique, PhD Thesis, Univ. of Paris VII, June 1992.

[8] R. Milner. “A Theory of Type Polymorphism in Programming Languages”, Journal of Computer and System Science 17, 1978.

[9] R. Milner, Communication and Concurrency, Prentice Hall, 1989.

[10] R. Milner, “The Polyadic π-calculus: a tutorial”. Technical Report ECS-LFCS-91-180, Dept. of Computer Science, Univ. of Edinburgh, Oct. 91.

[11] F. Nielson. “The typed λ-calculus with first-class processes”. In Proceedings of PARLE’89, LNCS Vol. 366, Springer-Verlag, 1989.

[12] H.R. Nielson, F. Nielson. “Higher-Order Concurrent Programs with Finite Communication Topology”. ACM Symposium on Principles of Programming Languages, 1994.

[13] D. Rémy, "Typechecking records and variants in a natural extension of ML", ACM Symposium on Principles of Programming Languages, 1989.

[14] D. Rémy, "Syntactic Theories and the Algebra of Record Terms", Research Report 1869, INRIA, March 93.

[15] J.H. Reppy, "CML: A Higher Order Concurrent Language", ACM SIGPLAN Conference on Programming Language Design and Implementation, Toronto, Canada, 1991.

[16] B. Thomsen. "A calculus of higher-order communicating systems", ACM Symposium on Principles of Programming Languages, 1989.

[17] B. Thomsen, "A second generation calculus for higher order processes", Acta informatica, 30:1–59, 1993.

[18] M. Tofte, "Operational Semantics and Polymorphic Type Inference", Ph.D. Thesis, Univ. of Edinburgh, May 1988.

[19] V. T. Vasconcelos, K. Honda, "Principal typing schemes in a polyadic π-calculus", In Proceedings of CONCUR'93, August 1993, LNCS Vol. 715, Springer-Verlag, 1993.

[20] M. Wand. "Complete Type Inference for Simple Objects", In Symposium on Logic in Computer Science, Ithaca, New York, june 1987. (A Corrigenda appeared in LICS 88.)

[21] M. Wand. "Type Inference for Objects with Instance Variables and Inheritance", to appear in Theoretical Aspects of Object-Oriented Programming, C. Gunter and J.C. Mitchell, eds, MIT Press. (Also Northeastern Univ. Technical Report NU-CCS-89-2)

APPENDIX – The unification algorithm

All instances of a tagged variable are assumed to occur with identically written indices; the variables in these indices not occurring elsewhere. This allows us to omit quantifiers in indices. Tagged variable symbols are assumed not overloaded at several sorts.

```
$\tau_1$ Unify $\tau_2 = \tau_1\ \mathbf{U}_\emptyset\ \tau_2$
where $\tau_1\ \mathbf{U}_\mathsf{L}\ \tau_2 = \text{case}\ \{\tau_1, \tau_2\}$ of
    $\{\alpha_1, \alpha_2\}$              $\Rightarrow$ if $\alpha_1 = \alpha_2$ then [ ] else $[\alpha_1/\alpha_2]$                (1)
    $\{\alpha, \tau\}$                    $\Rightarrow$ if $\alpha$ In $\tau$ then Fail else $[\tau/\alpha]$               (2)
    $\{\tau_1 \to \tau_2, \tau_1' \to \tau_2'\}$   $\Rightarrow$ (S $\tau_2\ \mathbf{U}_\emptyset$ S $\tau_2'$) o S where S = $\tau_1\ \mathbf{U}_\emptyset\ \tau_1'$   (3)
    $\{\Delta_1^{\overline{\sigma 1}}, \Delta_2^{\overline{\sigma 2}}\}$            $\Rightarrow$ if $\Delta_1^{\overline{\sigma 1}} = \Delta_2^{\overline{\sigma 2}}$ then [ ] else U o X1 o X2     (4)
                          where  U = if Monotype ($\overline{S\sigma 1}$)
                                     then $[\nabla^{(\overline{S}\overline{\sigma 1})}/\Delta_2^{\overline{\sigma 2}}]$ o $[\nabla^{(\overline{S}\overline{\sigma 1})}/\Delta_1^{\overline{\sigma 1}}]$
                                     else $[\Delta_1^{(\overline{S}\overline{\sigma 1})}/\Delta_2^{\overline{\sigma 2}}]$ o $[\Delta_1^{(\overline{S}\overline{\sigma 1})}/\Delta_1^{\overline{\sigma 1}}]$
                          and    X1 = Expand (L, $\Delta_1^{\overline{\sigma 1}}$)
                          and    X2 = Expand (L, $\Delta_2^{\overline{\sigma 2}}$)
                          where  S = $\sigma_1\ \mathbf{U}_\emptyset\ \sigma_2$
    $\{\Delta^{\overline{\sigma}}, \nabla^{\overline{\mu}}\}$               $\Rightarrow$ U o X                                         (5)
                          where  U = $[\nabla^{(\overline{S}\overline{\sigma})}/\Delta^{\overline{\sigma}}]$ o $[\nabla^{(\overline{S}\overline{\sigma 1})}/\Delta_1^{\overline{\sigma 1}}]$
                          and    X = Expand (L, $\Delta^{\overline{\sigma}}$)
                          where  S = $\sigma\ \mathbf{U}_\emptyset\ \mu$
    $\{\nabla^{\overline{\mu}}, \nabla^{\overline{\mu}}\}$               $\Rightarrow$ [ ]      where S = $\mu\ \mathbf{U}_\emptyset\ \mu'$               (6)
    $\{\{$p : $\tau\}\rho_1, \rho_2$ as $\Delta^\sigma\}$ or $\{\{$p : $\tau\}\rho_1, \rho_2$ as $\{$q : $\tau'\}\rho\}$        (7)
                          $\Rightarrow$ if p $\in$ L then $\rho_1\ \mathbf{U}_\mathsf{L}\ \rho_2$
                          else (S2 (S1 $\rho_1$) $\mathbf{U}_{\mathsf{L}\cup\{\mathsf{p}\}}$ S2 (S1 $\rho_2$)) o S2 o S1
                          where S2 = S1 $\tau\ \mathbf{U}_\emptyset\ \tau'$
                          where (S1 $\tau'$) = Extract (p, $\rho_2$)
    _                     $\Rightarrow$ if $\tau_1 = \tau_2$ then [ ] else Fail                 (8)
```

Each case matches the parameters of function $\mathbf{U}_L$ with some set pattern, e.g. both sets $\{\beta, \Psi\}$ and $\{\Psi, \beta\}$, for some type Ψ and plain variable β, match $\{\alpha, \tau\}$ in case (2). Algorithm **U** has a third parameter (its subscript) which is the set of labels for which its arguments have been unified so far (when these are tagged types). That set serves to skip hidden fields and to handle misordering of fields (in case (5)). The nonoverloading hypothesis for tagged variables allows to reuse one of the variable symbols in case (4).

Function **In** implements the occurrence check; it does not check occurrences of variables in hidden fields. Function **Expand** (L,$\Delta^{\overline{\sigma}}$) returns a substitution expanding $\Delta^{\overline{\sigma}}$ for all labels in L; expansions introduce copies of scheme σ in all fields, using fresh type variables. **Extract** (p,ρ) lookups the type associated with label p in ρ, expanding ρ on the fly if it has no such field; it returns that type and a substitution (empty or an expansion).

A Modal Analysis of Staged Computation

Rowan Davies and Frank Pfenning[*]
Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213, U.S.A.
`rowan@cs.cmu.edu and fp@s.cmu.edu`

Preliminary Version for the Workshop on
Types for Program Analysis,
Aarhus, Denmark,

May 1995

Abstract

We show that a type system based on the intuitionistic modal logic S4 provides an expressive framework for specifying and analyzing computation stages in the context of factional languages. Our main technical result is a conservative embedding of Nielson & Nielson's two-level factional language in our language Mini-ML$^{\Box}$, which in addition to partial evaluation also supports multiple computation stages, sharing of code across multiple stages, and run-time code generation.

[*]This work was sponsored in part by the Advanced Research Projects Agency, ARPA Order No. 8313.

1 Introduction

Dividing a computation into separate stages is a common informal technique in the derivation of algorithms. For example, instead of matching a string against a regular expression we may first compile a regular expression into a finite automaton and then execute the automaton on a given string. Partial evaluation divides the computation into two stages based on the early availability of some function arguments. Binding-time analysis determines which part of the computation may be carried out in a first (static) phase, and which part remains to be done in a second (dynamic) phase.

It often takes considerable ingenuity to write programs in such a way that they exhibit proper binding-time separation, that is, that *all* computation pertaining to the statically available arguments can in fact be carried out. From a programmer's point of view it is therefore desirable to declare the expected binding-time separation and obtain constructive feedback when the computation may not be staged as expected. This suggests that the binding-time properties of a function should be expressed in its types in a prescriptive type system, and that binding-time analysis should be a form of type checking. The work on two-level functional languages [NN92] and some work on partial evaluation (*e.g.* [GJ91, Hen91]) shows that this view is indeed possible and fruitful.

Up to now these type systems have been motivated *allgorithmically*, that is, they are explicitly designed to support partial evaluation. In this paper we show that they can also be motivated *logically*, and that the proper logical system for expressing computation stages is the intuitionistic variant of the modal logic S4. This observation immediately gives rise to a natural generalization of standard binding-time analysis by allowing multiple computation stages, sharing of code across multiple stages, and communication of binding-time information across module boundaries via types.

One of our conclusions is that when we extend the Curry-Howard isomorphism between proofs and programs from intuitionistic logic to the intuitionistic modal logic S4 we obtain a natural and logical explanation of computation stages. Each world in the Kripke semantics of modal logic corresponds to a stage in the computation. A term of type $\Box A$ corresponds to a piece of residual code to be executed in a future stage of the computation. The modal restrictions imposed on a type of the form $\Box A$ guarantee that a function of type $B \to \Box A$ can carry out *all* computation concerned with its first argument while generating the residual code of type A.

The starting points for our investigation are the systems for the intuitionistic modal logic S4 in [BdP92, PW95] and the two-level λ-calculus in [NN92]. We augment the former with recursion to obtain Mini-ML$^{\Box}$ and then show that a two-level functional language may be fully and faithfully embedded in Mini-ML$^{\Box}$. This verifies that Mini-ML$^{\Box}$ is indeed a conservative extension of the two-level language of [NN92] and thus correctly expresses standard binding-time separation. Following [PW95], we also sketch a compilation from Mini-ML$^{\Box}$ to a related language Mini-ML$^{\Box}_{e}$ whose operational semantics embodies the separation of evaluation into multiple stages.

2 Modal Mini-ML: An Explicit Formulation

This section presents Mini-ML$^{\Box}_{e}$, a language that combines some elements of Mini-ML [CDDK86] with a modal λ-calculus for intuitionistic S4, $\lambda_e^{\to\Box}$ [BdP92, PW95]. The presentation of the modal constructs differs from $\lambda_e^{\to\Box}$ in that we have a let form for de-constructing boxed values, and use two contexts in the typing rules. This avoids the need for syntactic substitutions, but does not alter the essential properties of the system.

For the sake of simplicity, we make the language explicitly typed, since we do not treat type inference here. We have also chosen not to include polymorphism, because there are issues regarding the interaction between type variables and computation stages that would distract from the main point of this paper.

2.1 Syntax

$$
\begin{array}{llrl}
\text{Types} & A & ::= & \mathsf{nat} \mid A_1 \to A_2 \mid A_1 \times A_2 \mid \Box A \\
\text{Terms} & E & ::= & x \mid \lambda x : A.\ E \mid E_1\ E_2 \\
 & & & \mid \mathbf{fix}\ x : A.\ E \mid \langle E_1, E_2 \rangle \mid \mathbf{fst}\ E \mid \mathbf{snd}\ E \\
 & & & \mid \mathbf{z} \mid \mathrm{s}\ E \mid (\mathbf{case}\ E_1\ \mathbf{of}\ \mathbf{z} \Rightarrow E_2 \mid \mathrm{s}\ x \Rightarrow E_3) \\
 & & & \mid \mathbf{box}\ E \mid \mathbf{let\ box}\ x = E_1\ \mathbf{in}\ E_2 \\
\text{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x : A
\end{array}
$$

We use A, B for types, Γ, Δ for contexts, and x for variables assuming that any variable can be declared at most once in a context. Bound variables may be renamed tacitly. We omit leading $\cdot$'s from contexts. We write $[E'/x]E$ for the result of substituting E' for x in E, renaming bound variables as necessary in order to avoid the capture of free variables in E'. The addition of types $\Box A$ to Mini-ML introduces two new term constructs: **box** E for introduction and **let box** $x = E_1$ **in** E_2 for elimination.

2.2 Typing Rules

Our typing rules for the Mini-ML fragment of the explicit language are completely standard. The problem of typing the modal fragment is well understood; we present here a variant of known systems [BdP92, PW95] inspired by zonal formulations of linear logic such as Girard's *LU* [Gir93]. Our typing judgment has two contexts, the first containing assumptions regarding all future worlds, and the second containing assumptions regarding the current world. Thus our judgement

$$\Delta; \Gamma \vdash^e \ E \ : \ A$$

would correspond to $\Box\Delta, \Gamma \vdash E^* : A$ in $\lambda_e^{\rightarrow\Box}$, where E^* is an appropriate obvious translation of E. Note that our system has the property that a valid term has a unique typing derivation. Due to space constraints, throughout this paper we have omitted the rules for **fix**, nat, and pairs, since they are completely standard. The rules are given in full in [DP95].

$$\frac{x : A \text{ in } \Gamma}{\Delta; \Gamma \vdash^e x : A}\ \mathsf{tpe\_lvar} \qquad \frac{x : A \text{ in } \Delta}{\Delta; \Gamma \vdash^e x : A}\ \mathsf{tpe\_gvar}$$

$$\frac{\Delta; \Gamma, x : A \ \vdash^e E : B}{\Delta; \Gamma \vdash^e \lambda x : A.\ E : A \rightarrow B}\ \mathsf{tpe\_lam} \qquad \frac{\Delta : \Gamma \vdash^e E_1 : A \rightarrow B \quad \Delta; \Gamma \vdash^e E_2 \ : \ A}{\Delta; \Gamma \vdash^e E_1\ E_2 : B}\ \mathsf{tpe\_app}$$

$$\frac{\Delta; \cdot \vdash^e E : A}{\Delta; \Gamma \vdash^e \mathbf{box}\ E : \Box A}\ \mathsf{tpe\_box} \qquad \frac{\Delta : \Gamma \vdash^e E_1 \ : \Box A \quad \Delta, x : A; \Gamma \vdash^e E_2 : B}{\Delta; \Gamma \vdash^e \mathbf{let\ box}\ x = E_1\ \mathbf{in}\ E_2 : B}\ \mathsf{tpe\_let\_box}$$

Note that the rule $\mathsf{tpe\_box}$ does not allow variables bound in the second context to appear in the body of a **box** constructor, and only the rule $\mathsf{tpe\_let\_box}$ binds variables in the first context.

2.3 Operational Semantics

The Mini-ML fragment of our system has a standard operational semantics. For the modal part, we interpret **box** E as a value containing the frozen computation E which may be carried out in a future stage. We evaluate **let box** $x = E_1$ **in** E_2 as a substitution of the residual code generated by E_1 for x in E_2 and then evaluating E_2. The residual code for E_1 will then be evaluated during the evaluation of E_2 as necessary.

Note that if $E : A$ and $E \hookrightarrow V$ then $V : A$ and V is unique. Mini-ML has this property, which is easy to establish by induction over the structure of an evaluation. Also note that we have omitted types in terms from the rules below, since they are irrelevant here.

$$\text{Values } V ::= \lambda x.\ E \mid \langle V_1, V_2\rangle \mid \mathrm{z} \mid \mathrm{s}\ V \mid \mathbf{box}\ E.$$

$$\frac{}{\lambda\ x.\ E \hookrightarrow \lambda\ x.\ E}\ \textsf{ev\_lam}$$

$$\frac{E_1 \hookrightarrow \lambda\ x.\ E_1' \qquad E_2 \hookrightarrow V_2 \qquad [V_2/x]E_1' \hookrightarrow V}{E_1\ E_2 \hookrightarrow V}\ \textsf{ev\_aap}$$

$$\frac{}{\mathbf{box}\ E \hookrightarrow \mathbf{box}\ E}\ \textsf{ev\_box} \qquad \frac{E_1 \hookrightarrow \mathbf{box}\ E_1' \quad [E_1'/x]E_2 \hookrightarrow V_2}{\mathbf{let\ box}\ x = E_1 \mathbf{in}\ E_2 \hookrightarrow V_2}\ \textsf{ev\_let\_box}$$

Note that in the evaluation of well-typed terms, only terms inside a box constructor are ever substituted into another box constructor.

2.4 Example: The Power Function in Explicit Form

We now show how we can define the power function in Mini-ML$_e^\Box$: in such a way that has type $\mathsf{nat} \to \Box(\mathsf{nat} \to \mathsf{nat})$, assuming a closed term *times* : $\mathsf{nat} \to \mathsf{nat} \to \mathsf{nat}$ (definable in the Mini-ML fragment in the standard way).

$$\begin{aligned}
\mathit{power} \equiv\ & \mathbf{fix}\ p : \mathsf{nat} \to \Box(\mathsf{nat} \to \mathsf{nat}). \\
& \quad \lambda n : \mathsf{nat}.\ \mathbf{case}\ n \\
& \qquad \mathbf{of}\ \mathrm{z}\ \Rightarrow \mathbf{box}\ (\lambda x : \mathsf{nat}.\ \mathrm{sz}) \\
& \qquad \mid\ \mathrm{s}\ m \Rightarrow \mathbf{letbox}\ q = p\ m\ \mathbf{in}\ \mathbf{box}\ (\lambda x : \mathsf{nat}.\ \mathit{times}\ x\ (qx))
\end{aligned}$$

The type $\mathsf{nat} \to \Box(\mathsf{nat} \to \mathsf{nat})$ expresses the that function evaluates everything that depends on the first argument of type nat (the exponent) and return residual code of type $\Box(\mathsf{nat} \to \mathsf{nat})$. Indeed, we calculate with our operational semantics:

$$\begin{array}{rl}
\mathit{power}\ \mathrm{z} \hookrightarrow & \mathbf{box}\ (\lambda x : \mathsf{nat}.\ \mathrm{s\ z}) \\
\mathit{power}\ (\mathrm{s\ z}) \hookrightarrow & \mathbf{box}\ (\lambda x : \mathsf{nat}.\ \mathit{times}\ x\ ((\lambda x : \mathsf{nat}.\ \mathrm{s\ z})x)) \\
\mathit{power}\ (\mathrm{s}\ (\mathrm{s\ z})) \hookrightarrow & \mathbf{box}\ (\lambda x : \mathsf{nat}.\ \mathit{times}\ x\ ((\lambda x : \mathsf{nat}.\ \mathit{times}\ x\ (\lambda x : \mathsf{nat}.\ \mathrm{s\ z})x))x))
\end{array}$$

Modulo some trivial redices of variables for variables, this is the result we would expect of partial evaluation.

2.5 Implementation Issues

The operational semantics of Mini-ML$_e^\Box$ may be implemented by a translation into pure Mini-ML, mapping $\Box A$ to $\mathsf{unit} \to A$; **box** E to $\lambda u : \mathsf{unit}.\ E$; and **let box** $x = E_1$ **in** E_2 to $(\lambda x' : \mathsf{unit} \to A.\ [x'()/x]E_2)E_1$. It may then appear that the modal fragment of Mini-ML$_e^\Box$ is redundant. Note, however, that the type $\mathsf{unit} \to A$. does not express any binding time properties, while $\Box A$ does. It is precisely this distinction which makes Mini-ML$_e^\Box$ interesting: the type checker will reject programs which may execute correctly, but for which the desired bindingtime separation is violated. Without the modal operator, this property cannot be expressed and consequently not checked.

Another implementation method would be to interpret $\Box A$ as a data-type representing code that calculates a value of type A. This code could be either machine code, source code, or some intermediate language. This would allow optimization after specialization, and could also support an operation to output code as a separate program. The representation must support substitution of one code fragment into another, as required by the ev_let_box rule. If the code is machine code, this naturally leads to the idea of templates, as used in run-time code generation (see [KEH93]). The deferred compilation approach in [LL94] would provide a more sophisticated implementation, supporting fast run-time generation of optimized code.

3 Modal Mini-ML: An Implicit Formulation

We now define an implicit version Mini-ML$^\Box$ of the explicit Mini-ML$_e^\Box$, following [PW95] where an implicit system $\lambda^{\to\Box}$ was defined. This system is more reasonable as a programming language, since we do not have to explicitly stage computation as required with **let box** $x = E_1$ **in** E_2. The operational semantics of the new system is given in terms of a type-preserving compiIation to the explicit system. Our development differs from [PW95] in that we introduce a term constructor **pop**. This means that typing derivations for valid terms are unique and the compilation from implicit to explicit terms is deterministic, avoiding some unpleasant problems concerning coherence.

3.1 Syntax

Types	A	::=	nat	$\mid A_1 \to A_2 \mid A_1 \times A_2 \mid \Box A$
Terms	M	::=	x	$\mid \lambda x : A.\ M \mid M_1\ M_2$
				$\mid \mathbf{fix}\ x : A.\ M \mid \langle M_1, M_2 \rangle \mid \mathbf{fst}\ M \mid \mathbf{snd}\ M$
				$\mid \mathbf{z} \mid \mathrm{s}\ M \mid (\mathbf{case}\ M_1\ \mathbf{of}\ \mathbf{z} \Rightarrow M_2 \mid \mathrm{s}\ x \Rightarrow M_3)$
				$\mid \mathbf{box}\ M \mid \mathbf{unbox}\ M \mid \mathbf{pop}\ M$
Contexts	Γ	::=	$\cdot$	$\mid \Gamma, x : A$
Context Stacks	Ψ	::=	$\cdot$	$\mid \Psi; \Gamma$

All the categories, except *context stacks* are standard. The importance of context stacks will be apparent when we present the typing rules.

3.2 Typing Rules

In this section we present typing rules for Mini-ML$^{\Box}$ using context stacks. The typing judgment has the form

$$\Psi; \Gamma \vdash^i M : A \quad \text{term } M \text{ has type } A \text{ in local context } \Gamma \text{ under stack } \Psi.$$

The context stack enables the distinguished use of variables depending on their relative position with respect to the **box** operators that enclose the term being typed. Intuitively, each element Γ of the context stack Ψ corresponds to a computation stage. The variables declared in Γ are the ones whose values will be available during the corresponding evaluation phase. When we encounter a term **box** M we enter a new evaluation stage, since M will be frozen during evaluation. In this new phase, we are not allowed to refer to variables of the prior phases, since they may not be available when M is unfrozen ("unboxed"). Thus, variables may only be looked up in the current (innermost) context (rule tpi_var) which is initialized as empty when we enter the scope of a **box** (rule tpi_box). However, *code* generated in the current or earlier stages may be used, which is represented by the rules tpi_unbox and tpi_pop.

$$\frac{x : A \text{ in } \Gamma}{\Psi; \Gamma \vdash^i\ x : A}\ \textsf{tpi\_var} \qquad \frac{\Psi; (\Gamma, x : A) \vdash^i\ M : B}{\Psi; \Gamma \vdash^i\ \lambda x : A.\ M : A \to B}\ \textsf{tpi\_lam}$$

$$\frac{\Psi; \Gamma \vdash^i\ M : A \to B \qquad \Psi; \Gamma \vdash^i\ N : A}{\Psi; \Gamma \vdash^i\ M\ N : B}\ \textsf{tpi\_app}$$

$$\frac{\Psi; \Gamma; \cdot \vdash^i\ M : A}{\Psi; \Gamma \vdash^i\ \textbf{box}\ M : \Box A}\ \textsf{tpi\_box} \qquad \frac{\Psi; \Gamma \vdash^i\ M : \Box A}{\Psi; \Gamma \vdash^i\ \textbf{unbox}\ M : A}\ \textsf{tpi\_unbox}$$

$$\frac{\Psi; \Delta \vdash^i\ M : \Box A}{\Psi; \Delta; \Gamma \vdash^i\ \textbf{pop}\ M : \Box A}\ \textsf{tpi\_pop}$$

Note that it may be useful to consider the modal fragment of the implicit language to be a statically typed analogue to the quoting mechanism in Lisp. Then **box** corresponds to backquote and **unbox** (**pop** ·) to comma. **unbox** alone corresponds to eval, while **pop** alone corresponds to quoting an expression generated with comma. Note however that our implementation via a compilation to Mini-ML$^{\Box}_{e}$ is quite different from Lisp quoting.

3.3 Examples in Implicit Form

We now show how we can define the power function in Mini-ML$^{\Box}$ in a simpler form than in Mini-ML$^{\Box}_{e}$, though still with type $\mathsf{nat} \to \Box(\mathsf{nat} \to \mathsf{nat})$. We use $\textbf{unbox}_i\ M$ as syntactic sugar for $\textbf{unbox}(\textbf{pop}^i\ M)$.

$$\begin{array}{l} \textit{power} \equiv \textbf{fix}\ p : \mathsf{nat} \to \Box(\mathsf{nat} \to \mathsf{nat}). \\ \qquad\qquad \lambda n : \mathsf{nat}.\ \textbf{case}\ n \\ \qquad\qquad\quad \textbf{of}\ \mathrm{z}\ \Rightarrow \textbf{box}\ (\lambda x : \mathsf{nat}.\ \mathrm{s\ z}) \\ \qquad\qquad\quad \mid \mathrm{s\ m} \Rightarrow \textbf{box}\ (\lambda x : \mathsf{nat}.\ \textit{times}\ x\ (\textbf{unbox}_1(p\ m)x)) \end{array}$$

As another example, we show how to define a function of type $\mathsf{nat} \to \Box\mathsf{nat}$ that returns a **box**'ed copy of its argument:

$$\begin{array}{lll} \textit{lift}_{\mathsf{nat}} \equiv \textbf{fix}\ f : \mathsf{nat} \to \Box(\mathsf{nat}.\ \lambda x : \mathsf{nat}.\textbf{case} & & x\ \textbf{of} \\ & & \mathrm{z} \Rightarrow \textbf{box}\ \mathrm{z} \\ & \mid & \mathrm{s}\ x' \Rightarrow \textbf{box}\ (\mathrm{s}\ (\textbf{unbox}_1\ (f\ x'))) \end{array}$$

A similar term of type $A \rightarrow \Box A$ that returns a **box** 'ed copy of its argument exists exactly when A is observable, *i.e.*, contains no $\rightarrow$. This justifies the inclusion of the *lift* primitive in two-level languages such as in [GJ91], and in fact in a more realistic version of our language it could also be included as a primitive.

3.4 Translation to Explicit Language

We do not define an operational semantics for Mini-ML$^{\Box}$ directly; instead we depend upon a translation to Mini-ML$^{\Box}_{e}$. This translation recursively extracts terms inside a **pop** constructor and binds the result of their evaluation to new variables, bound with a **let box** outside the enclosing **box** constructor. Variables thus bound occur exactly once.

The compilation from implicit to explicit terms is perhaps most easily understood if we restrict **pop** to occur only immediately underneath an **unbox** or another **pop**. On the pure fragment terms then follow the grammar

$$\begin{array}{llcll}
\text{Terms} & M & ::= & x & \mid \lambda x : A.\ M \mid M_1\ M_2 \\
 & & & & \mid \textbf{box}\ M\ \mid \textbf{unbox}\ P \\
\text{Pops} & P & ::= & M & \mid \textbf{pop}\ P
\end{array}$$

The extension to the full language including recursion is tedious but trivial. Any term can be transformed to one satisfying our restriction by replacing isolated occurrences of **pop** M by **box** (**unbox** (**pop** (**pop** M))).

The compilation below keeps track of the context in which the term to be translated should be placed (the k argument). This is necessary so that when we encounter an **pop** operator we can find the matching **box** operator and insert a **let box** binding in the resulting explicit term. We use the notation $k = \Lambda h.\ E$ for a context k with hole h. Filling the hole is written as an application $k(E')$. This must be implemented as syntactic replacement since k is intended to capture variables free in E'. First, the translation on terms, $[\![M]\!]\ k$.

$$\begin{array}{lcl}
[\![x]\!]\ k & = & k(x) \\
[\![M_1\ M_2]\!]\ k & = & [\![M_1]\!](\Lambda h_1.\ [\![M_2]\!](\Lambda h_2.\ (h_1\ h_2))) \\
[\![\lambda x.\ M]\!]\ k & = & [\![M]\!](\Lambda h.\ k(\Lambda x.\ h)) \\
[\![\textbf{box}\ M]\!]\ k & = & [\![M]\!](\Lambda h.\ k(\textbf{box}\ h)) \\
[\![\textbf{unbox}\ P]\!]\ k & = & [P]k(\Lambda h.\ h))
\end{array}$$

Nested **pop** operators are translated by traversing the current context k from the inside out until a **box** operator is found. This cancels one **pop** operator and continues the translation. After all **pop** operators have been removed (possibly none), we introduce a **let box** and continue the translation. The b argument accumulates the body of the **let box** which will eventually be introduced.

$$\begin{array}{lcl}
[\mathbf{pop}\ P](\Lambda h.\ k(E_1\ h))\ b & = & [\mathbf{pop}\ P]\ k(\Lambda h.\ E_1\ b(h)) \\
[\mathbf{pop}\ P](\Lambda h.\ k(h\ E_2))\ b & = & [\mathbf{pop}\ P]\ k(\Lambda h.\ b(h)\ E_2) \\
[\mathbf{pop}\ P](\Lambda h.\ k(\lambda x.\ h))\ b & = & [\mathbf{pop}\ P]\ k(\Lambda h.\lambda x.\ b(h)) \\
[\mathbf{pop}\ P](\Lambda h.\ k(\mathbf{box}\ h))\ b & = & [P]\ k(\Lambda h.\mathbf{box}\ b(h)) \\
\multicolumn{3}{l}{[\mathbf{pop}\ P](\Lambda h.\ k(\mathbf{let\ box}\ x = h\ \mathbf{in}\ E_2))\ b} \\
 & = & [\mathbf{pop}\ P]\ k(\Lambda h.\ \mathbf{let\ box}\ x = b(h)\ \mathbf{in}\ E_2) \\
\multicolumn{3}{l}{[\mathbf{pop}\ P](\Lambda h.\ k(\mathbf{let\ box}\ x = E_1\ \mathbf{in}\ h))\ b} \\
 & = & [\mathbf{pop}\ P]\ k(\Lambda h.\ \mathbf{let\ box}\ x = E_1\ \mathbf{in}\ b(h))
\end{array}$$

$$[M]\ k\ b = [\![M]\!](\Lambda h.\ k(\mathbf{let\ box}\ y = h\ \mathbf{in}\ b(y))) \quad \text{where } y \text{ is new}$$

Since h must occur exactly once in $\Lambda h.\ E$, the cases for $[\mathbf{pop}\ P]\ k\ b$ leave out only $\Lambda h.\ h$. If the original term is well-typed this case can never arise. An important invariant of $[P]\ k$ bis that $\Lambda h.\ k(b(h))$ remains the same in every recursive call. At present we have not formally proven that the translation above maps well-typed explicit terms to well-typed implicit terms. A related, slightly more complicated translation has been proven correct in [PW95].

As an example of this translation, it maps the above definition of *power* to the previous explicit one.

It is important to note that the operational semantics induced by the trans-lation is very different from the natural one defined directly on Mini-ML$^{\Box}$. In [MM94] a simple reduction semantics for a system similar to our implicit system is introduced which does not reflect binding time separation in any way. It is instead used to prove a Church-Rosser theorem and strong normalization for a pure modal λ-calculus.

4 A Two-level Language

In this section we define Mini-ML$_2$, a two-level functional language very close to the one described in [NN92]. We then define a simple translation into Mini-

ML$^\Box$ and prove that binding-time correctness in Mini-ML$_2$ is equivalent to modal correctness of the translation in Mini-ML$^\Box$.

Our language differs slightly from [NN92] in that we inject *all* run-time types into compile-time types, instead of just function types. This follows [GJ91], where there is no such restriction. Also, we find it convenient to divide the variables and contexts into run-time and compile-time, which involves a small change in the "up" and "down" rules. All other differences to [NN92] are due to minor differences between their underlying language and Mini-ML.

4.1 Syntax

$$
\begin{array}{llcrl}
\text{Run-time Types} & \tau & ::= & \underline{\mathsf{nat}} & \mid \tau_1 \underline{\rightarrow} \tau_2 \mid \tau_1 \underline{\times} \tau_2 \\
\text{Compile-time Types} & \sigma & ::= & \overline{\mathsf{nat}} & \mid \sigma_1 \overline{\rightarrow} \sigma_2 \mid \sigma_1 \overline{\times} \sigma_2 \mid \tau \\
\text{Terms} & e & ::= & \underline{x} & \mid \underline{\lambda x} : \tau.\ e \mid e_1 \mathbin{\underline{@}} e_2 \\
 & & & & \mid \underline{\mathbf{fix}}\ \underline{x} : r.\ e \mid \underline{\langle} e_1, e_2 \underline{\rangle} \mid \underline{\mathbf{fst}}\ e \mid \underline{\mathbf{snd}}\ e \\
 & & & & \mid \underline{\mathbf{z}} \mid \underline{\mathrm{s}}\ e \mid (\underline{\mathbf{case}}\ e_1\ \underline{\mathbf{of}}\ \underline{\mathrm{z}} \Rightarrow e_2 \mid \underline{\mathrm{s}}\ \underline{x} \Rightarrow e_3) \\
 & & & \mid \overline{y} & \mid \overline{\lambda y} : \sigma.\ e \mid e_1 \mathbin{\underline{@}} e_2 \\
 & & & & \mid \overline{\mathbf{fix}}\ \overline{y} : \sigma.\ e \mid \overline{\langle} e_1, e_2 \overline{\rangle} \mid \overline{\mathbf{fst}}\ e \mid \overline{\mathbf{snd}}\ e \\
 & & & & \mid \overline{\mathbf{z}} \mid \overline{\mathrm{s}}\ e \mid \overline{(\mathbf{case}}\ e_1 \overline{\mathbf{of}}\ \overline{\mathbf{z}} \Rightarrow e_2 \mid \overline{\mathrm{s}}\ \overline{y} \Rightarrow e_3) \\
\text{Run-time Contexts} & \Gamma & ::= & \cdot & \mid \Gamma, \underline{x} : \tau \\
\text{Compile-time Contexts} & \Delta & ::= & \cdot & \mid \Delta, \overline{y} : \sigma
\end{array}
$$

4.2 Typing Rules

Run-time Typing

$$
\frac{\underline{x} : \tau \text{ in } \Gamma}{\Delta; \Gamma \vdash^r\ \underline{x} : \tau}\ \mathsf{tpr\_var}
\qquad
\frac{\Delta \vdash^c\ e : \tau}{\Delta; \Gamma \vdash^r\ e : \tau}\ \mathsf{down}
$$

$$
\frac{\Delta; \Gamma; \underline{x} : r_2 \vdash^r\ e : \tau}{\Delta; \Gamma \vdash^r\ \underline{\lambda x} : r_2.\ e : \tau_2 \underline{\rightarrow} \tau}\ \mathsf{tpr\_lam}
\qquad
\frac{\Delta; \Gamma \vdash^r\ e_1 : \tau_2 \underline{\rightarrow} \tau \qquad \Delta; \Gamma \vdash^r e_2 : \tau_2}{\Delta; \Gamma \vdash^r\ e_1 \mathbin{\underline{@}} e_2 : \tau}\ \mathsf{tpr\_app}
$$

Compile-time Typing

$$\frac{\overline{y} : \sigma \text{ in } \Delta}{\Delta; \Gamma \vdash^c \ \overline{y} : \sigma}\ \textsf{tpc\_var} \qquad \frac{\Delta; \cdot \vdash^r \ e : \tau}{\Delta \vdash^c \ e : \tau}\ \textsf{up}$$

$$\frac{\Delta, \overline{y} : \sigma_2 \vdash^c \ e : \sigma}{\Delta \vdash^c \ \overline{\lambda} \overline{y} : \sigma_2.\ e : \sigma_2 \overline{\rightarrow} \sigma}\ \textsf{tpc\_lam} \qquad \frac{\Delta \vdash^c \ e_1 : \sigma_2 \overline{\rightarrow} \sigma \quad \Delta \vdash^c e_2 : \sigma_2}{\Delta \vdash^c \ e_1 \overline{@}\ e_2 : \sigma}\ \textsf{tpc\_app}$$

Note that we remove run-time assumptions at the down rule, while in [NN92] this is done later at the up rule. This change is justified since by the structure of their rules, such assumptions can never be used in the compile-time deduction in between.

4.3 Translation to Implicit Modal Mini-ML

The translation to Mini-ML$^\Box$ is now very simple. We translate both run-time and compile-time Mini-ML fragments directly, and insert $\Box$, **box**, **unbox** and **pop** to represent the changes between phases. We define two mutually recursive functions to do this: $\|\cdot\|$ is the run-time translation and $|\cdot|$ is the compile-time translation. We overload this notation between types and terms. We write $\underline{e}$ and $\overline{e}$ to match any term whose top constructor matches the phase annotation.

Run – time Types

$$\begin{array}{rcl} ||\underline{\textsf{nat}}|| & = & \textsf{nat} \\ ||\tau_1 \underline{\rightarrow} \tau_2|| & = & ||\tau_1|| \rightarrow ||\tau_2|| \\ ||\tau_1 \underline{\times} \tau_2|| & = & ||\tau_1|| \times ||\tau_2|| \end{array}$$

Compile – time Types

$$\begin{array}{rcl} |\overline{\textsf{nat}}| & = & \textsf{nat} \\ |\sigma_1 \overline{\rightarrow} \sigma_2| & = & |\sigma_1| \rightarrow |\sigma_2| \\ |\sigma_1 \overline{\times} \sigma_2| & = & |\sigma_1| \times |\sigma_2| \\ |\tau| & = & \Box||\tau|| \end{array}$$

Run – time Terms

$$\begin{array}{rcl} ||\underline{x}|| & = & x \\ ||\underline{\lambda x} : \tau.\ e|| & = & \lambda x : ||\tau||.\ ||e|| \\ ||e_1 \ \underline{@}\ e_2|| & = & ||e_1||\ ||e_2|| \\ ||\overline{e}|| & = & \textbf{unbox}\ (\textbf{pop}|\overline{e}|) \end{array}$$

Compile − time Terms

$$\begin{array}{rcl} |\overline{y}| & = & y \\ |\overline{\lambda} \overline{y} : \tau.\ e| & = & \lambda y : |\tau|.\ |e| \\ |e_1 \overline{@}\ e_2| & = & |e_1|\ |e_2| \\ |\underline{e}| & = & \mathbf{box}\ ||\underline{e}|| \end{array}$$

4.4 Equivalence of Binding Time Correctness and Modal Correctness

In this section we state our main theorem, which is that binding time correctness is equivalent to modal correctness of the translation to Mini-ML$^\Box$.

Theorem 1

1. *If* $||e|| = M$ *then:*
   (a) *if* $\Delta; \Gamma \vdash^r e : \tau$ *then we have* $|\Delta|; ||\Gamma|| \vdash^i M : ||\tau||$*;*
   (b) *if* $|\Delta|; ||\Gamma|| \vdash^i M : A$ *then we have* $|\Delta|; ||\Gamma|| \vdash^r e : \tau$ *with* $||\tau|| = A$*.*
2. *If* $|e| = M$ *then:*
   (a) *if* $\Delta \vdash^c e : \sigma$ *then we have* $|\Delta| \vdash^i M : |\sigma|$*;*
   (b) *if* $|\Delta| \vdash^i M : A$ *then we have* $\Delta \vdash^c e : \sigma$ *with* $|\sigma| = A$*.*

Proof: By simultaneous induction on the definitions of $||e||$ and $|e|$. Note that we can take advantage of strong inversion properties, since we have exactly one typing rule for each term constructor in Mini-ML$^\Box$ and Mini-ML$_2$, plus the up and down rules to connect the $\vdash^c$ and $\vdash^r$ judgements.[1]
$\Box$

By examining this proof we can verify that the translation of a two-level term can always be type-checked only using the tpi_unbox and tpi_pop rules when tpi_un box immediately follows tpi_pop. This corresponds to a weaker modal logic, K, in which we drop the assumption in S4 that the accessibility relation is reflexive and transitive [MM94].

[1]See [DP95] for proof details.

In fact, we can define a language Mini-ML$^{\Box}_{K}$ by replacing the **unbox** and **pop** constructors with one equivalent to $\mathbf{unbox}_1$ as in [MM94]. Then, Mini-ML$^{\Box}_{K}$ closely models Mini-ML$_2$, but permits an arbitrary number of phases, each of which can only execute the code generated by the immediately preceding one. This is similar to the idea of B-level languages in [NN92] (with B linearly ordered), and in fact a B-level version of Mini-ML would be exactly equivalent to Mini-ML$^{\Box}_{K}$, by a natural extension of the two-level translation. It is also similar to the Multi-level Generating Extensions of [GJ95].

It is interesting then to consider what the reflexitivity and transitivity assumptions model in the context of staged computation. Essentially they allow us to execute generated code at any future time, or immediately. It would be difficult to achieve the same in an extension of a two-level language, since the separation between the levels is achieved by duplicating the term and type constructors. Hence we consider Mini-ML$^{\Box}$ to be an appropriate language in which to study more general forms of staged computation, including run-time code generation.

5 Extended Example

In [GJ95] the calculation of inner products is given as an example of a program with more than two phases. We now show how this example can be coded in Mini-ML$^{\Box}$. Note that we have assumed a data type vector in the example, along with a function $sub : \mathsf{nat} \rightarrow \mathsf{vector} \rightarrow \mathsf{nat}$ to access the elements of a vector. We also use $\mathbf{let}\ x = E_1\ \mathbf{in}\ E_2$ as syntactic sugar for $(\lambda x : A.\ E_2)E_1$.

Then, the inner product example without staging is expressed in Mini-ML as follows:

```
let $iprod$ = fix $ip : \mathsf{nat} \rightarrow \mathsf{vector} \rightarrow \mathsf{vector} \rightarrow \mathsf{nat}.$
              $\lambda n : \mathsf{nat}.$ case $n$
                of $\mathrm{z} \Rightarrow \lambda v : \mathsf{vector}.\ \lambda w : \mathsf{vector}.\ \mathrm{z}$
                | $\mathrm{s}\ n' \Rightarrow \lambda v : \mathsf{vector}.\ \lambda w : \mathsf{vector}.$
                                  $plus\ (times\ (sub\ n\ v)\ (sub\ n\ w))\ (ip\ n'\ v\ w)$
in ...
```

We add in $\Box$, **box** and $\mathbf{unbox}_i$ to get a function with three computation stages. We assume a function $lift_{\mathsf{nat}}$ as defined earlier and a function sub' :

$\mathsf{nat} \rightarrow \Box(\mathsf{vector} \rightarrow \mathsf{nat})$ which is a specializing version of *sub*, that perhaps pre-computes some pointer arithmetic based on the array index. We first define a staged version *times′* of *times* which avoids the multiplication in the specialization if the first argument is zero. This will speed up application of *iprod′* to its third argument, particularly in the case that the second argument is a sparse vector.

```
let times′ : □(nat → □(nat → nat)) =
            box (λm : nat. case m
              of z ⇒ box (λn : nat. z)
              | s m′ ⇒ box (λn : nat. times n (unbox₁ (mathitlift_nat m))))
in let iprod′ = fix ip : nat → □(vector → □(vector → nat)).
              λn : nat. case n
                    of z ⇒ box (λv : vector. box (λw : vector. z))
                    | s n′ ⇒ box (λv : vector. box (λw : vector.
                       plus (unbox₁ (unbox₁ times′(unbox₁(sub′ n) v))
                                    (unbox₂(sub′ n) w))
                            (unbox₁ (unbox₁(ip n′) v) w)))
in let iprod3 : □(vector → □(vector → nat)) = iprod′ 3.
in let iprod3a : □(vector → nat) = unbox iprod3 [7, 0, 9].
in let iprod3b : □(vector → nat) = unbox iprod3 [7, 8, 0].
in ...
```

The last four lines show how to execute the result of a specialization using **unbox** without **pop** (corresponding to eval in Lisp). Also, the occurence of $\mathbf{unbox}_2$ indicates code used at the third stage but generated at the first. These two aspects could not be expressed within a multi-level language.

Note the erasure of the $\mathbf{unbox}_i$ and **box** constructors in *iprod′* leaves *iprod*, except that we used a different version of multiplication. The operational semantics of the two programs is of course quite different.

We have also experimented with programming some other standard examples from partial evaluation in Mini-ML$^\Box$, including regular expression matching and the Ackerman function, both of which are easily expressible.

6 Conclusion and Future Work

In this paper we have proposed a logical interpretation of binding times and staged computation in terms of the intuitionistic modal logic S4. We first presented an explicit language Mini-ML$_e^\Box$ (including recursion, natural numbers, and pairs) and its natural operational semantics. This language is too verbose to be practical, so we continued by defining an implicit language Mini-ML$^\Box$ which, with some syntactic sugar, might serve as the core for an extension of a language with the complexity of Standard ML. The operational semantics of Mini-ML$^\Box$ is given by a compilation to Mini-ML$_e^\Box$. It generalizes Nielson & Nielson's two-level functional language [NN92] which is demonstrated by a conservative embedding theorem, the main technical result of this paper.

The two-level language we consider, Mini-ML$_2$, is directly based on the one in [NN92], but has a stricter binding-time correctness criterion than used, for example, in [GJ91]. Essentially, this restriction may be traced to the fact that our underlying evaluation model applies only to closed terms, while [GJ91] seems to require evaluation of terms with free variables. Glück and Jørgensen [GJ95] present a multi-level binding-time analysis with the less strict binding-time correctness criterion, along with practical motivations for multi-level partial evaluation, though they do not treat higher order functions. A modal operator similar to the "next" operator from temporal logic looks promising as a candidate to model this looser correctness criterion, but we have yet to develop this line of research.

Our language Mini-ML$^\Box$ requires the insertion of the **box**, **unbox** and **pop** coercions into a functional program. It may be preferable for these coercions to remain implicit, though in such a language valid expressions no longer have unique or even principal types, thus raising coherence problems. We intend to study a language in which the modal types are considered refinements of the usual Mini-ML types, using intersections to express principal types (see [FP91] for analogous non-modal refinement types). Refinement type inference for this language would be a form of generalized, polyvariant binding-time analysis. Compilation would be type-directed, generating different versions of functions appropriate for different stagings of computation. The programmer would control this process through refinement type constraints imposed upon functions by type annotations. Type inference in such a language would need to depend strongly on subtyping via implicit coercions between refinement types.

Our operational semantics is also rather naive from a partial evaluation point of view. In particular, we do not memoize during specialization. A memoizing semantics would be desirable for a serious implementation, and would require some restrictions on side-effects. See [BW93] for a description of a serious partial evaluator for Standard ML, which in part inspired this work.

This paper does not treat polymorphism, though it seems that it should not cause any problems. We conjecture that computational effects can essentially be treated as in other work on partial evaluation by prohibiting, through typing, that effects move between computation stages (see, for example, [BW93]). We expect our type system to interact very well with ML's module system. In fact, part of our motivation was to provide the programmer with means to specify staging (= binding time) information in a signature and thus propagate it beyond module boundaries.

Our approach provides a general logically motivated framework for staged computation that includes aspects of both partial evaluation and run-time code generation. As such it should allow efficient code to be generated within a more declarative style of programming, and provides an automatic check that the intended staging is achieved. We have implemented a simple version of Mini-ML$^{\Box}$ in the logic programming language Elf [Pfe91]. To date we have only experimented with small examples, but plan to carry out more realistic experiments in the near future.

7 Acknowledgements

We gratefully acknowledge discussions with Lars Birkedal, Olivier Danvy, Joëlle Despeyroux, Andrzej Filinski, Karoline Malmkjær, Greg Morrisett, Morten Welinder, and Hao-Chi Wong regarding the subject of this paper.

References

[BdP92] Gavin Bierman and Valeria de Paiva. Intuitionistic necessity revisited. In *Proceedings of the Logic at Work Conference*, Amsterdam, Holland, December 1992.

[BW93] Lars Birkedal and Morten Welinder. Partial evaluation of Standard ML. Technical Report DIKU-report 93/22, DIKU, Department of Computer Science, University of Copenhagen, October 1993.

[CDDK86] Dominique Clément, Joëlle Despeyroux, Thierry Despeyroux, and Gilles Kahn. A simple applicative language: Mini-ML. In *Proceedings of the 1986 Conference on LISP and Functional Programming*, pages 13–27. ACM Press, 1986.

[DP95] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. Technical Report CMU-CS-95-145, Carnegie Mellon University, Department of Computer Science, May 1995.

[FP91] Tim Freeman and Frank Pfenning. Refinement types for ML. In *Proceedings of the SIGPLAN '91 Symposium on Language Design and Implementation, Toronto, Ontario*, pages 268–277. ACM Press, June 1991.

[Gir93] Jean-Yves Girard. On the unity of logic. *Annals of Pure and Applied Logic*, 59:201–217, 1993.

[GJ91] Carsten Gomard and Neil Jones. A partial evaluator for the untyped lambda-calculus. *Journal of Functional Programming*, 1(1):21–69, January 1991.

[GJ95] Robert Glück and Jesper Jørgensen. Efficient multi-level generating extensions. Unpublished Manuscript, 1995.

[Hen91] Fritz Henglein. Efficient type inference for higher-order binding-time analysis. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture, 5th ACM Conference,* volume 523 of *Lecture Notes in Computer Science*, pages 448–472. Springer, Berlin, Heidelberg, New York, 1991.

[KEH93] David Keppel, Susan J. Eggers, and Robert R. Henry. A case for runtime code generation. Technical Report Technical Report 93-11-02, Department of Computer Science and Engineering, University of Washington, November 1993.

[LL94] Mark Leone and Peter Lee. Deferred compilation: The automation of run-time code generation. In *Proceedings of the Workshop on Partial Evaluation and Semantics-based Program Manipulation (PEPM'94), Orlando*, June 1994. An earlier version appears as Carnegie Mellon School of Computer Science Technical Report CMU-CS-93-225, November 1993.

[MM94] Simone Martini and Andrea Masini. A computational interpretation of modal proofs. In H. Wansing, editor, *Proof Theory of Modal Logics.* Kluwer, 1994. Workshop proceedings, To appear.

[NN92] Flemming Nielson and Hanne Riis Nielson. *Two-level Functional Languages.* Cambridge University Press, 1992.

[Pfe91] Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

[PW95] Frank Pfenning and HaoChi Wong. On a modal λ-calculus for S4. In S. Brookes and M. Main, editors, *Proceedings of the Eleventh Conference on Mathematical Foundations of Programming Sematics*, New Orleans, Louisiana, March 1995. To appear in *Electronic Notes in Theoretical Computer Science*, Volume 1, Elsevier.

Optimizing lazy functional programs using flow inference

Karl-Filip Faxén[*]

Dept. of Teleinformatics, Royal Institute of Technology, Electrum 204, S-164 40 Kista, tel: +46 8 752 13 78, fax: +46 8 751 17 93

kff@it.kth.se

Abstract

Nonstrict higher order functional programruing languages are notorious for their low run time efficiency. Optimizations based on *flow analysis*, which determines for each variable x in a program which expressions could have originated the value of x, can improve the situation by removing redundant eval and thunk operations, avoiding thunk updates, and allowing the use of unboxed representations of some data. We formulate flow analysis as an inference problem in a type system built using type inclusion constraints and an algorithm for solving these constraints is also given.

1 Introduction

Polymorphically typed nonstrict higher order functional programming languages are a boon to the programmer because they provide powerful mechanisms for abstraction [11]. Equally, and for the same reasons, they are very difficult to compile to efficient code.

Among the main obstacles are the frequent need to build thunks (representations for unevaluated expressions), test whether objects are thunks or

[*]Work partly funded by Esprit BRA 8130 LOMAPS

WHNFs (i.e. integers, cons-cells, partial applications, etc), call statically unknown code (in order to evaluate a thunk or in a higher order function), and update thunks with the result of their evaluation (to preserve sharing).

The interaction of polymorphism, thunks and garbage collection also creates problems by forcing compilers to use a uniform, one-size-fits-all representation of all data regardless of type. Since some objects, e.g. Cons cells and thunks, must be boxed (represented as pointers into the heap), *all* objects are forced to be boxed.

Part of the problem is solved by *strictness analysis* [15, 10] which can reduce the number of thunks that need to be built, creating opportunities to eliminate evals (the operations that test if an object is a thunk and if so evaluates it) and ultimately allow the use of specialized representations (like unboxed integers etc). But strictness analysis in itself does not exploit these additional opportunities — it only allows us to eliminate some of the thunks.

Instead, we use *flow analysis* to exploit these opportunities. By flow analysis, we mean an analysis that determines, for each variable x in a program P, a safe approximation to the set of expressions that might have *originated* the value of x. This information can be post-processed in simple ways to yield information that allows a compiler to eliminate evals and thunks, use unboxed data representations, omit thunk updates and optimize calls to statically unknown code.

We analyze programs in Fleet, a Functional Language with Explicit Evals and Thunks, intended as a compiler intermediate language. Variable binding, function application, etc is strict, but there are thunk and eval expressions that can be used to express lazy evaluation explicitly. Strictness analysis can be used prior to flow analysis to generate as few thunk expressions as possible.

Possible *originators* in Fleet are lambda abstractions, constructor and operator applications, and thunk expresions. Every originator is labeled with a unique *originator label* and every object is tagged with the label of the expression that originated it.

We give Fleet a semantics in the style of natural semantics by defining a rewriting relation between *configurations* (triples of a store, an environments and an expression) and *results* (pairs of a store and an address in the store).

For each variable x in a Fleet program there is a set $\overline{x}$ of labels, called the *tag set* of x, such that a run-time type error occurs, ie the execution "goes wrong", if x is ever bound to an object whose label is not in $\overline{x}$.

Type inference in the system presented in Section 3 either fails or derives values for the $\overline{x}_i$ such that the execution of the program is guaranteed not to "go wrong". Any program that is well-typed in the Hindley-Milner system can be translated to a Fleet program that is well-typed in the flow type system.

The flow type system is formulated in terms of type inclusion constraints and an algorithm for solving these is given.

We discuss three applications of the analysis; eval/thunk elimination, unboxing (which we generalize to representation selection), and avoidance of unnecessary updates by sharing analysis.

Finally, we present preliminary experimental results for some small benchmarks which show that the optimizations we have discussed can cut execution time with a factor ranging from about 1.6 to about 3.2.

1.1 A note on notation

We will use standard function notation; given a function f, $f[x \mapsto y]$ is a function that maps x to y and otherwise behaves as f. We will also write *id* for the identity function.

2 Fleet

The syntax and semantics of Fleet is presented in Figs. 1 and 2, respectively, and an example program is given in Fig. 3. We give Fleet a *natural semantics*, similar to those given in [12] and [19], which fairly closely models a modern graph reduction implementation of a lazy functional language. In particular, we model the graph explicitly, so environments map variables to *adresses* which are mapped to *closures* by the graph.

$$a \in \text{Addr} \qquad \rho \in \text{Var} \rightarrow \text{Addr} \qquad G \in \text{Addr} \rightarrow \text{Closure}$$

A closure is a pair of an environment and a value (a lambda abstraction, con-structor application or thunk expression). The label of a closure (ρ, e), written *label*(ρ, e), is the label of e. The rules in Fig. 2 allow us to prove sentences of the form $\langle G, \rho, e\rangle \Downarrow \langle G', a\rangle$ which are to be read "in graph G and environment ρ the expression e rewrites to the address a in the graph G'". A program P is a closed expression and the result of executing P is $\langle G, a\rangle$ iff $\langle [\,], [\,], P\rangle \Downarrow \langle G, a\rangle$.

$$
\begin{array}{rcl}
e \in \text{Expr} & ::= & x \mid x_1\, x_2 \mid l\ \ \texttt{\textbackslash}x\ \texttt{->}\ e \\
& \mid & l\ \ C\, x_1 \ldots x_r \mid l\ \ op\, x_1 \ldots x_r \\
& \mid & \texttt{case}\ x\ \texttt{of}\ alt_1; \ldots; alt_n\ \texttt{end} \\
& \mid & \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \\
& \mid & l\ \texttt{thunk}\ e \mid \texttt{eval}\ x \\
alt \in \text{Alt} & ::= & C\ x_1 \ldots x_r\ \texttt{->}\ e
\end{array}
$$

$C \in \text{Constr} ::= \texttt{Nil} \mid \texttt{Cons} \mid \ldots \mid \texttt{0} \mid \texttt{1} \mid \ldots \qquad op \in \text{Op} ::= \texttt{inc} \mid \texttt{add} \mid \ldots$

$x \in \text{Var} \qquad l \in \text{Lab} ::= \texttt{@1} \mid \texttt{@2} \mid \ldots \qquad \overline{x} \in \mathcal{P}(\text{Lab}) ::= \{\updownarrow_\infty, \ldots, \updownarrow_{||}\}$

We will often write c for either a constructor (C) or an operator (op). A *value* is a lambda abstraction, a constructor application, or a thunk expression. An *originator* is a value or an operator application. We call the set of lables that label thunk and non-thunk originators LabT and LabW, respectively.

Figure 1: Syntax of Fleet

In an expression `let` $x = e_1$ `in` e_2, x *must* occur tree in e_2 and if x occurs free in e_1, then e_1 has to be a value (this makes the rule for let expressions in Fig. 2 unproblematic; if x occurs free in e_1 then e_1 is simply put in a closure whose environment part refers to the closure itself).

The only cases when existing closures are overwritten are when thunks are evaluated in the second rule for `eval` in Fig. 2. Here, the thunk to be evaluated is first overwritten with a black hole and subsequently with an indirection to the result of its evaluation. In both cases the new closure is a thunk with the same label as the original. This means that the label of a closure at a certain address never changes.

If, when rewriting $\langle G, \rho, e\rangle$, some variable x in e is bound to a closure labeled with a label not in $\overline{x}$, then $\langle G, \rho, e\rangle \Downarrow \langle G, \textit{wrong}\rangle$ where *wrong* is a distinguished address that is used to signal a run-time type error.

There is no implicit evaluation of thunks in those contexts where a whnf

$$\langle G,\ \rho,\ x\rangle \Downarrow \langle G,\ \rho(x)\rangle$$

$$\frac{G(\rho(x_1)) = (\rho',\ l \setminus x\ \ \texttt{->}\ \ e) \qquad \langle G,\ \rho'[x \mapsto \rho(x_2)],\ e\rangle \Downarrow \langle G',\ a\rangle}{\langle G,\ \rho,\ x_1\ x_2\rangle \Downarrow \langle G',\ a\rangle}$$

$$\frac{a \notin \mathit{Dom}(G) \qquad e \text{ is a value}}{\langle G,\ \rho,\ e\rangle \Downarrow \langle G[a \mapsto (\rho, e)],\ a\rangle}$$

$$\frac{a \notin \mathit{Dom}(G) \quad G(\rho(x_1)) = ([\,], l_1\, C_1)\ \ldots\ G(\rho(x_k)) = ([\,], l_k\, C_k) \quad [\![\mathit{op}]\!](C_1, \ldots, C_k) = C}{\langle G,\ \rho,\ l\ \mathit{op}\ x_1 \ldots x_k\rangle \Downarrow \langle G[a \mapsto ([\,],\ l\ C)],\ a\rangle}$$

$$\frac{G(\rho(x)) = (\rho',\ l\ C\ x'_1 \ldots x'_k) \qquad \langle G,\ \rho[x_1 \mapsto \rho'(x'_1), \ldots, x_k \mapsto \rho'(x'_k)],\ e\rangle \Downarrow \langle G',\ a\rangle}{\langle G,\ \rho,\ \texttt{case}\ x\ \texttt{of}\ \ \ldots;\ C\ x_1 \ldots x_k\ \texttt{->}\ e;\ \ldots\ \ \texttt{end}\rangle \Downarrow \langle G',\ a\rangle}$$

$$\frac{\langle G,\ \rho[x \mapsto a],\ e_1\rangle \Downarrow \langle G',\ a\rangle \qquad \langle G',\ \rho[x \mapsto a],\ e_2\rangle \Downarrow \langle G'',\ a'\rangle}{\langle G,\ \rho,\ \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2\rangle \Downarrow \langle G'',\ a'\rangle}$$

$$\frac{G(\rho(x)) = (\rho',\ l\ \texttt{thunk}\ e) \qquad \langle G[\rho(x) \mapsto ([\,],\ l\ \texttt{thunk}\ \bullet)],\ \rho',\ e\rangle \Downarrow \langle G',\ a'\rangle}{\langle G,\ \rho,\ \texttt{eval}\ x\rangle \Downarrow \langle G'[\rho(x) \mapsto ([\mathtt{r} \mapsto a'],\ l\ \texttt{thunk}\ \mathtt{r})],\ a'\rangle}$$

$$\frac{G(\rho(x)) = (\rho', e) \text{ and } e \text{ is not a thunk}}{\langle G,\ \rho,\ \texttt{eval}\ x\rangle \Downarrow \langle G,\ \rho(x)\rangle}$$

$$\frac{\mathit{label}(G(\rho(x))) \notin \overline{x} \text{ or } \rho(x) = \mathit{wrong} \text{ for some } x \in \mathit{Dom}(\rho)}{\langle G,\ \rho,\ e\rangle \Downarrow \langle G, \mathit{wrong}\rangle}$$

The meaning $[\![\mathit{op}]\!]$ of a k-ary operator *op* is a partial function from Constr^k to Constr, e.g. $[\![\texttt{inc}]\!]\ 3 = 4$ and so on. Some operations, e.g. divide by zero, are undefined.

Figure 2: Rewriting expressions

is needed; instead, an explicit eval operation has to be used, as the one on line 3 in Figure 3.

3 The type system

In ordinary Hindley-Milner style type systems [14, 4], types are generally either function types or algebraic datatypes (e.g. lists). Such types correspond to *raw types* in Fleet. To the raw types, we add *annotated types* and *label*

```
let from = @1 \ n -> let rest = @2 thunk                             -- 1
                              let inc_n = @3 thunk                   -- 2
                                             let ne = eval n         -- 3
                                             in @4 inc ne            -- 4
                              in from inc_n                          -- 5
                     in @5 Cons n rest                               -- 6
in let zero = @6 0                                                   -- 7
   in from zero                                                      -- 8
```

Figure 3: An example: Printing the natural numbers

types to capture information about labels[1]. Note that raw types can contain nested annotated types and label types.

$$\begin{array}{rcl} \tau & \in & \text{AnnoType} ::= \langle \eta, \nu \rangle \mid v \\ \eta & \in & \text{LabelType} ::= \{l_1, \ldots, l_n\} \mid \; w \mid \; w \cap \{l_1, \ldots, l_n\} \\ \nu & \in & \text{RawType} ::= \tau_1 \texttt{ -> } \tau_2 \mid \; T\ \eta_1 \ldots \eta_k\ \tau_1 \ldots \tau_n \mid \; u \\ v & \in & \text{AVar} \quad w \in \text{LVar} \cup \text{Var} \quad u \in \text{RVar} \end{array}$$

Here, AVar, LVar and RVar are three disjoint sets of type variables, and the use of program variables ($\in$ Var) in LabelTypes will be explained below. Sometime it will be convenient to allow Ω to range over any kind of type expression and α over any kind of type variable (including program variables):

$$\Omega \in \text{TypeExp} ::= \tau \mid \eta \mid \nu \qquad \alpha \in \text{TypeVar} ::= v \;\mid\; w \;\mid\; u$$

Figure 4 gives some example expressions and their types. In these examples

`@6 0`	:	`⟨{@6}, Int⟩`
`@3 Cons (@1 17) (@2 Nil)`	:	`⟨{@3}, List {@2} ⟨{@1}, Int⟩⟩`
`@1 \ x -> @2 inc x`	:	`⟨{@1}, ⟨LabW, Int⟩ -> ⟨{@2}, Int⟩⟩`
`@1 thunk @2 inc (@3 17)`	:	`⟨{@1,@2}, Int⟩`

Note that we have ignored the requirement that all arguments to operators have to be variables, but the diligent reader can insert the reqired let-expressions.

Figure 4: Examples of expressions and their types

[1]Annotated types play a similar role to the decorated types in [23].

we have given very precise types to expressions. For instance, the integer constructor expression @6 `0` in the first example was given the type $\langle\{@6\},\ \texttt{Int}\rangle$ but can also be typed as $\langle\{@3, @6\},\ \texttt{Int}\rangle$. Intuitively, this type is less precise than $\langle\{@6\},\ \texttt{Int}\rangle$ because there are more expressions producing integers labeled with @3 or @6 than there are expression producing integers labeled only with @6. We formalize this intuition as an ordering $\sqsubseteq$ on type expressions, defined in Figure 5 (this is essentially a subtype ordering). We now have $\langle\{@6\},\ \texttt{Int}\rangle \sqsubseteq \langle\{@3, @6\},\ \texttt{Int}\rangle$.

$$\begin{array}{rcll}
\tau_1 \texttt{ -> } \tau_2 & \sqsubseteq & \tau_1' \texttt{ -> } \tau_2' & \Leftrightarrow \tau_1' \sqsubseteq \tau_1 \wedge \tau_2 \sqsubseteq \tau_2' \\
T\ \Omega_1 \dots \Omega_k & \sqsubseteq & T\ \Omega'_1 \dots \Omega'_k & \Leftrightarrow \Omega_1 \sqsubseteq \Omega_1' \wedge \dots \wedge \Omega_k \sqsubseteq \Omega_k' \\
\langle \eta,\ \nu \rangle & \sqsubseteq & \langle \eta',\ \nu' \rangle & \Leftrightarrow \eta \sqsubseteq \eta' \wedge \nu \sqsubseteq \nu' \\
\{l_1, \dots, l_n\} & \sqsubseteq & \{l_1', \dots, l_{n'}'\} & \Leftrightarrow \{l_1, \dots, l_n\} \subseteq \{l_1', \dots, l_{n'}'\} \\
 & & \alpha \sqsubseteq \alpha &
\end{array}$$

Figure 5: The $\sqsubseteq$-ordering

We define a *substitution* θ to be a function of the form $id[\alpha_1 \mapsto \Omega_1, \dots, \alpha_n \mapsto \Omega_n]$ mapping type and program variables to type expressions in such a way that AVars are mapped to AnnoTypes and so on (program variables are mapped to LabelTypes). We also extend substitutions to type expressions in the obvious way.

A *type inclusion constraint* is an inequality of the form $\Omega \leq \Omega'$ and is *solved* by a substitution θ iff $\theta(\Omega) \sqsubseteq \theta(\Omega')$.[2] A set S of type incluzon constraints is solved by θ if every constraint in S is solved by θ, and we write $Sol(S)$ for the set of all θ that solves S. We also sometimes write $\tau \leq_\pi x$ where x is a program variable as a shorthand for $\tau \leq \langle x, u\rangle$ where u is some RVar not occurring anywhere else in the derivation. Constraints of this form are called *tag constraints* and will be used to determine safe values for the tag sets $\overline{x}_i$.

Like in all Hindley-Milner style type systems, we use *type schemes* to express polymorphic types. The syntax of type schemes is given by

$$\sigma \in \text{ TypeScheme } ::= \tau \mid \forall \alpha_1 \dots \alpha_k \ .\ \tau \ \textit{where}\ S$$

[2]In inclusjon bmed type inference systems, the symbol $\subseteq$ is generally used where we use $\sqsubseteq$ and $\leq$, in addition to its use for set inclusion, but we prefer to keep these symbols separate.

where S is a set of type inclusion constraints and we quantify over LVars, Rvars and AVars but never over Vars. Such a type scheme stands for all types τ' such that there is a substitution $\theta = id[\alpha_1 \mapsto \Omega_1, \ldots, \alpha_k \mapsto \Omega_k]$, such that $Sol(\theta(S)) \neq \emptyset$ and $\tau' = \theta(\tau)$. Hence, τ is equivalent to $\forall\ .\ \tau\ where\ \{\}$. We extend substitutions to type schemes as follows

$$\theta(\forall \alpha_1 \ldots \alpha_k\ .\ \tau\ where\ S) = \forall\ \alpha_1 \ldots \alpha_k\ .\ \theta(\tau)\ where\ \theta(S)$$

where we assume that no name clashes will occur (for every quantified variable α_i, that $\theta(\alpha_i) = \alpha_i$ and for no other $\alpha \in Dom(\theta)$, does α_i occur in $\theta(\alpha)$).

3.1 Type inference

In a type inference system, one infers the type of an expression from assumptions about the types of the free variables of the expression. For instance, in the Hindley-Milner system one would infer that $f\ x$ has type τ' from the assumption that f has type τ -> τ' and x has type τ, or symbolically, $\{f\ :\ \tau \text{ -> } \tau',\ x\ :\ \tau\} \vdash f\ x\ :\ \tau'$. The relation between the types of f, x and $f\ x$ is expressed by τ and τ' occurring both in the type of f and in the types of x and $f\ x$, respectively.

An alternate approach would be to express this relation explicitly with a separate set of constraints, written between the assumptions and $\vdash$: $\{f\ :\ \tau'',\ x\ :\ \tau\},\ \{\tau'' = \tau \text{ -> } \tau'\} \vdash\ f\ x\ :\ \tau'$. As a generalization, one can use inclusion constraints rather than equality constraints; then we have $\{\tau'' \subseteq \tau\text{->}\ \tau'\}$ in place of the above constraint. See Aiken *et. al.* [1] and Smith [21] for type systems defined in this way.

In the flow type system, our example inference looks as follows:

$$\{f\ :\ \tau'',\ x\ :\ \tau\},\ \{\tau'' \leq \langle \eta, \tau \text{ -> } \tau' \rangle\} \vdash\ f\ x\ :\ \tau'$$

As can be seen from the inference rules in Figure 6, typing judgements in our system are of the form $A,\ S \vdash e\ :\ \tau$, where A is a set of typing assumptions of the form $x\ :\ \sigma$, S is a set of type inclusion constraints, e an expression and τ an AnnoType. The ALT rule forms a slight exception to this in that it has the form $A,\ S, \nu \vdash\ C\ x_1 \ldots x_k \text{ -> }\ e\ :\ \tau$. Here, the RawType ν is the type of the object we attempt to match $C\ x_1 \ldots x_k$ against.

There is one rule for each form of expression and there are no separate rules for instantiation and elimination. Instead, instantiation is merged with the VAR rule and generalization is merged with the LET rule. These rules use the auxilliary predicates $\mathcal{I}$ and $\mathcal{G}$, respectively. Studying their definitions, given in Fig. 7, we see that we never generalize over Vars. This restriction is the mechanism with which we combine flow information from the various uses of a let-bound variable. Note also the tag constraints in the ABS, ALT and LET rules.

Another thing to note about the LET rule is that, in contrast to systems like those in [1] and [21], we do not have the restriction that $\mathit{Sol}(S)$ be nonempty — instead we have the restriction that x *must* occur in e' (the body of the let expression).

Studying the THUNK and EVAL rules, we see that the only thing that happens, type-wise, when we suspend or restart computation is the addition or deletion, respectively, of some labels. Hence, if a function can accept a thunk as argument, then it can also accept the result of evaluating this thunk.

The inference rules OPCON and ALT also make use of constructor and operator typings. Each constructor C is defined by a *constructor typing* (which can be derived from some algebraic datatype declaration in the source program) of the form

$$C\ \tau_1 \ldots \tau_r\ :\ T\ w_1 \ldots w_k\ v_1 \ldots v_n$$

where only the type variables that occur to the right of : are allowed to occur to the left. Example constructor typings are:

$$\texttt{Cons v}\ \langle\texttt{w, List w v}\rangle\ :\ \texttt{List w v}\quad \text{and}\quad \texttt{Nil}\ :\ \texttt{List w v}.$$

Similarly, for each operator *op* there is an *operator typing* of an analogous form. An example is the typing of the `inc` operator, `inc` $\langle$LabW, `Int`$\rangle$: `Int`, which means that if `inc` is given an integer in whnf, the result is an integer.

3.2 Soundness

We define soundness as the following property: For every program P, set of type inclusion constraints S, and AnnoType τ, if $\emptyset,\ S \vdash\ P\ :\ \tau$ and *Sol(S)* $\neq \emptyset$ and $\langle [\,], [\,], P\rangle \Downarrow \langle G, a\rangle$ then $a \neq \mathit{wrong}$.

[*VAR*]

$$\frac{\mathcal{I}(\sigma,\ \mathcal{S},\ \tau)}{A \cup \{x : \sigma\}, S \cup S' \ \vdash\ x : \tau}$$

[*APP*]

$$\frac{A, S\ \vdash\ x_1 : \tau_1 \qquad A, S\ \vdash\ x_2 : \tau_2}{A, S \cup \{\tau_1 \leq \langle \text{LabW},\ \tau_2 \rightarrow \tau_3\rangle\}\ \vdash\ x_1\, x_2 : \tau_3}$$

[*ABS*]

$$\frac{A \cup \{x : \tau_1\}, S\ \vdash\ e : \tau_2}{A, S \cup \{\tau_1 \leq_\pi x\}\ \vdash\ l\ \backslash\, x\ \texttt{->}\ e\ :\ \langle \{l\},\ \tau_1 \rightarrow \tau_2\rangle}$$

[*OPCON*]

$$\frac{c\,\tau'_1 \ldots \tau'_r : \nu \qquad A, S\ \vdash\ x_1 : \tau_1 \quad \ldots \quad A, S\ \vdash\ x_r : \tau_r}{A, S \cup \{\tau_1 \leq \theta(\tau'_1), \ldots, \tau_r \leq \theta(\tau'_r)\}\ \vdash\ l\ \ c\, x_1 \ldots x_r : \langle \{l\},\ \theta(\nu)\rangle}$$

[*CASE*]

$$\frac{A, S\ \vdash\ x : \tau' \qquad A, S, \nu\ \vdash\ alt_1 : \tau_1 \quad \ldots \quad A, S, \nu\ \vdash\ alt_n : \tau_n}{A, S \cup \{\tau_1 \leq \tau, \ldots, \tau_n \leq \tau, \tau' \leq \langle \text{LabW},\ \nu\rangle\}\ \vdash\ \texttt{case}\ x\ \texttt{of}\ alt_1; \ldots; alt_n\ \texttt{end}\ : \tau}$$

[*ALT*]

$$\frac{C\ \tau'_1 \ldots \tau'_r : \nu' \qquad A \cup \{x_1 : \theta(\tau'_1), \ldots, x_r : \theta(\tau'_r)\}, S\ \vdash\ e : \tau}{A, S \cup \{\theta(\tau'_1) \leq_\pi x_1, \ldots, \theta(\tau'_r) \leq_\pi x_r, \nu \leq \theta(\nu')\}, \nu\ \vdash\ C\ x_1 \ldots x_r\ \texttt{->}\ e\ :\ \tau}$$

[*LET*]

$$\frac{A \cup \{x : \tau\}, S\ \vdash\ e : \tau' \quad \mathcal{G}(\mathcal{A}, \mathcal{S} \cup \{\tau' \leq \tau, \tau' \leq_\pi \S\}, \tau', \sigma) \quad \mathcal{A} \cup \{\S : \sigma\}, \mathcal{S}'\ \vdash\ ]' : \tau''}{A, S'\ \vdash\ \texttt{let}\ \ x\ \texttt{=}\ e\ \texttt{in}\ \ e'\ :\ \tau''}$$

[*THUNK*]

$$\frac{A, S\ \vdash\ e : \tau}{A, S \cup \{\tau \leq \langle \text{LabW},\ \nu\rangle, \tau \leq \langle \eta,\ \nu\rangle, \{l\} \leq \eta\}\ \vdash\ l\ \texttt{thunk}\ e\ :\ \langle \eta,\ \nu\rangle}$$

[*EVAL*]

$$\frac{A, S\ \vdash\ x : \tau}{A, S \cup \{\tau \leq \langle \eta,\ \nu\rangle\}\ \vdash\ \texttt{eval}\ x\ :\ \langle \eta \cap \text{LabW},\ \nu\rangle}$$

Figure 6: Type inference rules for Fleet

$$\mathcal{I}(\forall\,\alpha_{\infty}\ldots\alpha_{\|}\;.\;\tau \sqsupseteq\langle\rceil\nabla\rceil\;\mathcal{S},\;\theta(\mathcal{S}),\;\theta(\tau)) \Leftrightarrow \theta = id[\alpha_1 \mapsto \Omega_1,\ldots,\alpha_k \mapsto \Omega_k]$$

where the Ω_i are arbitrary type expressions such that θ is well formed

$$\mathcal{G}(\mathcal{A},\;\mathcal{S},\;\tau,\;\forall\,\alpha_{\infty}\ldots\alpha_{\|}\;.\;\tau \sqsupseteq\langle\rceil\nabla\rceil\;\mathcal{S}) \Leftrightarrow \{\alpha_{\infty},\ldots,\;\alpha_{\|}\} \cap (\mathcal{FVS}(\mathcal{A}) \cup \mathrm{Var}) = \emptyset$$

Figure 7: Generalization and instantiation

We have proved the soundness of this type system with respect to the operational semantics of Fleet. The proof is by induction on the height of the proof of the evaluation relation, and is similar in spirit to the one given by Leroy in [13] where he proves the soundness of the Hindley-Milner type system relative to a natural semantics of an extension to ML with references and continuations. Unfortunately, the proof is rather large, so we do not include it in this paper.

4 A Type Inference Algorithm

In this section we present an implementation of the analysis in the form of a type inference algorithm. Given a program (closed expression) P, the algorithm either fails if P is not type correct or, if P is type correct, finds values for the tag sets of the variables in P such that P will not go wrong.

The first step is to find the most general S and τ such that $\emptyset,\ S \vdash P\ :\ \tau$. It is rather straight forward to turn the inference rules in Fig. 6 into a recursive procedure that takes a set of typing assumptions A and an expression e as arguments and gives a set of inclusion constraints S and a type τ as result. Fresh type variables are used for the type expressions that occurs in a rule without occurring in the typing assumptions of the conclusion of the rule or in the type part of any of the premises.

Most implementations of lazy functional laguages pass the value of the program to the run-time system for printing (or interpretation, as in the case of the HASKELL I/O model). Given that we have obtained S and τ by the algorithm sketched above, we model I/O by the constraint $\tau \leq \tau_{run}$ where τ_{run} is an AnnoType representing the run-time system. Let $S_1 = S \cup \{\tau \leq \tau_{run}\}$.

We now simplify S_1 by applying the rules in Fig. 8 exhaustively. It is easy to see that the decomposition rules preserve solutions exactly since they correspond very closely to the definition of $\sqsubseteq$ in Fig. 5, The substitution rules preserve solutions in the sense that if $S \to \theta(S)$ by some substitution rule, then every solution θ' to S can be written as $\theta'' \circ \theta$ where $\theta'' \in Sol(\theta(S))$. This is because $\Omega \sqsubseteq \Omega'$ only is possible if Ω and Ω' have the same outermost type constructor. If this simplification process terminates then either all RawTypes and AnnoTypes remaining are variables, or the system has no solutions.

To see that simplification always terminates, call a constraint $\Omega \leq \Omega'$ where either Ω or Ω' is a type variable a *variable* constraint and call other constraints *nonvariable* constraints. Now, each application of a decomposition rule will eliminate one nonvariable constraint of height k and add some new constraints of height strictly less than k, for some k. Further, each application of a substitution rule will strictly decrease the number of variable constraints of height at most k without increasing the number of nonvariable constraints of height larger than k, for some k.

Let S_2 be the result of this simplification and suppose that all RawTyes and AnnoTypes remaining are variables; then we can instantiate all of the RVars and all of the AVars to an arbitrary RVar or AVar, respectively, and drop the resulting trivial constraints.

We now have a system S_3 of inequalities of the form $\eta_1 \leq \eta_2$. We solve these in three steps as follows:

1. Split every label set inclusion into one inclusion over LabT and one over LabW, as follows

$$\eta_1 \leq \eta_2 \to \eta_1 \cap \text{LabT} \leq \eta_2 \cap \text{LabT}, \quad \eta_1 \cap \text{LabW} \leq \eta_2 \cap \text{LabW}$$

   and simplify all set expressions as much as possible. There are now no LabelTypes of the form w left, only of the forms $w\cap$ LabW and $w\cap$ LabT. Call the result S_4

2. Let S_5 be the transitive closure of S_4.[3]

[3]The transitive closure of a set S of inequalities is the smallest set S^+ such that $S \subseteq S^+$ and $\{\eta \leq \eta',\ \eta' \leq \eta''\} \subseteq S^+ \Rightarrow \eta \leq \eta'' \in S^+$.

3. If there is some $\{l\} \leq \{l_1, \ldots, l_k\}$ such that $l \notin \{l_1, \ldots, l_k\}$ in S_5, type checking has failed. Otherwise, the smallest values of the $\overline{x}_i$ are given by:

$$\overline{x} = \{l \mid \{l\} \leq x \cap \mathrm{LabW} \in S_5 \text{ or } \{l\} \leq x \cap \mathrm{LabT} \in S_5\}$$

5 Applications of the analysis

In the following we assume that we have performed the analysis on a program (closed expression) and that for every x in the program, $\overline{x}$ is a safe tag set for x.

It turns out that many of the algorithms below can be formulated as reachability in a directed or undirected graph. This is good since there are fast algorithms for that problem.

5.1 Elimination of evals and thunks

It is clear that, in a Fleet-program, an expression of the form `eval` x can be replaced by x alone if no label that labels a `thunk`-expression is related to x.

It is also clear that if evaluation of the body e of a thunk-expression is certain to terminate (without runtime error) the thunk can be replaced by e (we call this *inlining* the thunk), even if it is not known if its value will be needed in whnf. Thus e gets executed *speculatively.* Inlining is advantageous if either e is very cheap to evaluate (a few machine instructions), or very likely to be evaluated later anyhow.

Now, these two observations are actually related since an eval is typically an expensive operation (in general, it is not even certain that it terminates) Hence the removal of an `eval` might enable the removal of a `thunk` which enables removal of further `eval`s and so on. In a recursive function, we might even have circular dependencies.

For instance, in the example in Figure 3, the thunk labelled @3 contains an application of the `inc` operator (a cheap, safe operation) and an eval of the variable `n`. Looking at the tag set of `n`, which is {@3, @4, @6}, we see that the only thunk that `n` can be bound to is @3. Hence, if we inline the

Decomposition and elimination rules:

$$\{\langle\eta_1,\ \nu_1\rangle \leq \langle\eta_2,\ \nu_2\rangle\} \cup S \rightarrow \{\eta_1 \leq \eta_2, \nu_1 \leq \nu_2\} \cup S$$

$$\{\tau_1 \texttt{ -> } \tau_2 \leq \tau_3 \texttt{ -> } \tau_4\} \cup S \rightarrow \{\tau_3 \leq \tau_1, \tau_2 \leq \tau_4\} \cup S$$

$$\{T\ \Omega_1 \ldots \Omega_k \leq T\ \Omega'_1 \ldots \Omega'_k\} \cup S \rightarrow \{\Omega_1 \leq \Omega'_1, \ldots, \Omega_k \leq \Omega'_k\} \cup S$$

$$\{\alpha \leq \alpha\} \cup S \rightarrow S$$

Substitution rules:

$S \rightarrow id[\alpha \mapsto \alpha_1 \texttt{ -> } \alpha_2](S)$ if $\alpha \leq \tau_1 \texttt{ -> } \tau_2$ (or vice versa) $\in S$

$S \rightarrow id[\alpha \mapsto T\ \alpha_1 \ldots \alpha_k](S)$ if $\alpha \leq T\ \Omega_1 \ldots \Omega_k$ (or vice versa) $\in S$

$S \rightarrow id[\alpha \mapsto \langle\alpha_1,\ \alpha_2\rangle](S)$ if $\alpha \leq \langle\eta,\ \nu\rangle$ (or vice versa) $\in S$

where the α_i are type variables not occurring in S

A substitution rule is only applicable if the substituted-for variable is not a proper subexpression of any Ω occurring in S.

Figure 8: Simplification rules

@3 thunk, `n` can only be bound to whnfs, so we can eliminate the eval and we arrive at the program in Figure 9.

```
let from = @1 \ n -> let rest = @2 thunk                   -- line 1
                                   let inc_n = @4 inc n  -- line 2
                                   in from inc_n         -- line 3
                        in @5 Cons n rest                 -- line 4
in let zero = @6 0                                        -- line 5
   in from zero                                           -- line 6
```

Figure 9: The example transformed

This leads to the following rules for eval/thunk elimination: Let R be the set of labels of *residual* thunks, i.e. those that should not be inlined, and let $R_0 \subseteq R$ be those thunks that should not be inlined even if all evals were eliminated (because they contain other unsafe or expensive operations). Let $\varepsilon\ : \text{Lab} \rightarrow \mathcal{P}(\text{Lab})$ be a function such that for each thunk expression l `thunk` e in the program, if X is the set of variables that are arguments to exposed evals in e (i.e. those not inside other thunk expressions in e), then $\varepsilon(l) = \cup\{\overline{x} \mid x \in X\}$ (intuitively; the evaluation of l might lead directly to the evaluation of any thunk with label in $\varepsilon(l)$. Then R must satisfy the following condition:

$$\text{If } \varepsilon(l) \cap R \neq \emptyset, \text{ then } l \in R$$

We can compute the smallest R satisfying the above by the following algorithm:

1. Let G be a directed graph whose nodes are the labels of all thunks and where there is an edge (l_1, l_2) iff $l_2 \in \varepsilon(l_1)$.
2. Let $R = \{l \mid \exists l' \in R_0 \ : \ l' \text{ is reachable from } l \text{ in } G\}$.

We can now inline every l `thunk` e such that $l \notin R$ and eliminate every `eval` x such that $\overline{x} \cap R = \emptyset$. We also adjust all of the $\overline{x}_i$ by removing the eliminated thunks, to enable later analyses to take advantage of the removal of the thunks.

5.2 Representation analysis

We will treat unboxing in this more general context since there are many special representations that one might want to use.

Let R be a set of *representations* and let $\hat{r} \in R$ be the *canonical representation* (for example R might be {Boxed, UnboxedInt, UnboxedFloat} with $\hat{r}$ = Boxed). As the example suggests, not all representations are possible for all data; a Cons cell can't be represented as an UnboxedFloat, and in addition not all operations may be supported for all representations; x can't be an UnboxedInt if it is the argument of an eval. We formalize this by writing $\Delta(l)$ for the set of possible representations of objects with label l and $\Delta(x)$ for variables. Further, we require $\hat{r}$ to be a possible representation for all originators and variables.

Since a representation is a static property of a variable, it is the same for every invocation of a function. Thus we can trace variables from activation records at garbage collection time since we can find static information about the activation record from the return address.

Now, representation selection is the problem of finding an assignment δ : Var $\cup$ Lab $\rightarrow R$ of representations to variables and originators such that $\delta(l) \in \Delta(l)$ and $\delta(x) \in \Delta(x)$ for all l and x, satisfying the following constraint:

$$l \in \overline{x} \Rightarrow \delta(l) = \delta(x)$$

This gives us the following algorithm for representation selection:

1. Let G be an undirected graph whose nodes are the variables and labels and where there is an edge $\{l, x\}$ iff $l \in \overline{x}$.

2. Let $G_1, \ldots, G_n$ be the connected components of G.

3. For each G_i:

    (a) Choose an $r \in \bigcap\{\Delta(z) \mid z \in G_i\}$.

    (b) For every $z \in G_i,\ \delta(z) = r$.

5.3 Sharing analysis

Sharing analysis answers two related questions: For an originator, it tells whether objects computed by this originator may be accessed more than once, and for a variable it tells whether any object the variable might be bound to may be accessed more than once. This information is important since it may allow us to destructively update or immediately garbage collect unshared objects.

We can compute sharing information from a syntactic characterization of variables together with flow information.

Consider the life of a heap cell allocated during computation. When it is born, it will be unshared since the storage allocator will return precisely one reference to it, to which precisely one variable will be bound. In order for this cell to become shared later, the reference must either be duplicated or stored in a heap cell that becomes shared.

So one way of looking at sharing analysis is to determine that references to an object are never duplicated and never stored in a shared closure in the graph. The first condition is quite simple: A variable that is *flow linear* (i.e. occurs at most once along any path of control in its scope) can't duplicate any references.

References gets captured in environments stored in the heap (or abstractly, in the graph) when rewriting values. In the case of lambda abstractions and constructor applications, they can then be read from the heap an arbitrary number of times, but in the case of thunks, however, the free variables will only be accessed once (if they are flow linear, of course) since the thunk is overwritten with a black hole at the start of its evaluation.

This leads to the following sharing conditions: Let S be the set of labels of possibly shared objects. Then the following consistency conditions must hold:

1. If $l \in \overline{x}$ and x is not flow-linear, then $l \in S$.
2. For each constructor application $l\ C\ x_1 \dots x_k$: If $l \in S$ then $\overline{x_i} \subseteq S$ for each x_i.
3. For each lambda abstraction l `\` x `->` e: If $l \in S$ then $\overline{y} \subseteq S$ for each y in $FV(e) - \{x\}$.

We can find the smallest S satisfying the above as follows:

1. Let G be a directed graph whose nodes are labels and where there is an edge (l_1, l_2) iff either
   (a) l_2 is the label of a constructor application of the form $l_2\ C\ x_1 \dots x_k$ and $l_1 \in \overline{x_i}$ for some x_i, or
   (b) l_2 is the label of a lambda abstraction l_2 `\` x `->` e and $l_1 \in \overline{y}$ for some $y \in FV(e) - \{x\}$.
2. Let $N = \bigcup\{\overline{x} \mid x \text{ is a flow-nonlinear variable}\}$
3. Then $S = \{l \mid \exists l' \in N \ :\ l' \text{ is reachable from } l \text{ in } G\}$.

6 Measurements

We have implemented flow inference in an experimental compiler for a simple lazy functional language called Plain. The Plain compiler contains a very simple backwards strictness analyzer which assumes all unknown functions to be non-strict, so it doesn't find much strictness in programs using a lot of higher order functions.

The compiler implements flow inference as described in this paper (except for an improved constraint solution algorithm) and uses the results to perform eval/thunk-elimination, representation selection and update avoidance. The analysis can be turned off for comparison, in which case the generated code is similar in performance to the code generated by the Chalmers LML-compiler with optimization turned on (_O to lmlc).

We have measured the effectiveness of the optimizations on the following small test programs:

`qh` The N-Queens program written with lots of higher order functions; even length and append are defined in terms of foldr Run with N=10.

`q1` Same as `qh`, but with all higher order functions specialized. The transformation from `qh` to `q1` could be done automatically by a compiler, but we have done the transformation by hand.

`qf` Same as `q1`, but some intermediate lists eliminated by deforestration [24]. `append` Writes two copies of stdin on stdout. Run with a 92KByte textfile as input.

`nrev` Naive reverse of a 1KB textfile.

`nfib` The not-really-Fibonnaci function that counts the number of function calls. Run with input 32.

The Plain compiler can generate instrumentation code that counts various kinds of events, including eval checks, thunk constructions and calls, updates, and boxing and unboxing operations. In Table 1 we summarize the memurements we have made of the test programs. For each measured quantity we give the number of occurrences in the unoptimized code and the percentage of those occurrences that were eliminated by the optimizations. Thus 100% means that there were (almost) no occurrences of that event in the optimized code. The timing measurements were made on a lightly loaded 40 MHz SPARCstation 10 with 32 MBytes of memory and no second level cache running SunOS 4.1.3. The times reported are CPU times.

7 Related work

What we call flow analysis is known by a number of different names. It has been called *closure analysis* by Sestoft [18] and others, *flow information* by Steckler and Wand [25] and others, and *control flow information* by Jones, Shivers [20], and others.

Program	eval		trunk		box		unbox	
	#	%	#	%	#	%	#	%
`qh`	9587	78	1660	47	372	6	5330	45
`ql`	8579	95	1241	66	349	100	5330	100
`qf`	7023	99	822	95	384	100	5030	100
`append`	554	50	185	0	462	100	647	100
`nrev`	1361	50	678	0	5	100	6	100
`nfib`	7049	100	0	14	10574	100	14098	100

Program	updates		time (seconds)		
	#	%	unopt	opt	speedup
`qh`	2987	92(54)	4.69	2.90	1.62
`ql`	2175	100(100)	3.75	1.75	2.14
`qf`	1362	100(100)	2.88	1.08	2.67
`append`	370	50(0)	1.92	1.06	1.81
`nrev`	1356	100(100)	4.42	1.40	3.16
`nfib`	0	65(18)	6.04	1.92	3.15

Table 1: Event count and timing measurements (all counts in thousands)

The way in which the information is computed also varies. In older work, such as that by Sestoft, Jones and Shivers, *abstract interpretation* is generally used, but recently there have been a number of formulations in terms of constraint systems, among which are [25] and Heintze [8]. Also, Palsberg [16] has used a similar framework. Common to these analyses is that the analysis problem to be solved is set up in terms of the expressions etc in the program to be analyzed. Therefore, the entire program is always needed, and separate compilation becomes very troublesome. Analyses based on type and effect inference have been studied since in a type inference setting one only need the types of the free identifiers in the program. An example is [22], where Tang and Jouvelot study a combination of effect inference and abstract interpretation (they use abtract interpretation for its superior accuracy as compared to their monomorphic type system).

When it comes to exploitation of the analysis information, most of the above use the information for replacing function parameters with global variables or binding time analysis for partial evaluation. Sestoft , in [18], however, explores the use of closure information to turn first order analyses into higher order analyses which is close in spirit to our applications were we combine flow information with local information.

Recently, Boquist [3] has applied a constructor analysis due to Johnsson to the optimization of a graph reduction intermediate language. The transformations performed are related to our eval elimination, but the formulation is rather different. Boquist also uses the analysis information to enable interprocedural register allocation.

The applications we discuss have also been much studied separately, at least from the theoretical point of view,

Thunk inlining is already discussed by Mycroft (but only for a first order language) [15] and by Augustsson [2] who also discusses eval elimination, although he does not give any algorithm and does not discuss the interaction. Gomard and Sestoft [6, 7] and others have studied *path analysis* which is similar to eval elimination.

Boxing has been treated by several authors, a recent example being [9] where Henglein and Jørgensen defines an optimality criterion for boxing completions in a call-by-value language. Here boxing is used only because of polymorphism, but they do not treat the requirements from garbage collection and lazy evaluation. Their approach is considarably different from ours in that it is not based on program analysis, but rather on stepwise, meaning preserving transformations, and it is also more general than our approach.

On the other hand, Peyton-Jones and Launchbury [17] take account of all the requirements (polymorphism, gc and lazy evaluation), but do not give any algorithm but rather argues that representation selection should be done by the programmer.

In [5], Goldberg presents an abstract interpretation that detects sharing of partial applications in a nonstrict higher order language. He also applies it to the generation of fully lazy supercombinators. Also, Hughes discusses a backwards sharing analysis in [10].

References

[1] Alexander Aiken, Edward L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *Conference Record of POPL '94: 21st*

ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 163–173, Portland, Oregon, January 1994.

[2] Lennart Augustsson. *Compiling Lucy Functional Languages, Part II.* PhD thesis, Chalmers University of Technology, 1987.

[3] Urban Boquist. Interprocedural register allocation for lazy functional languages. In *FPCA'95*, 1995.

[4] Luis Damas and Bobin Milner. Principal type-schemes for functional programs. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[5] Benjamin Goldberg. Detecting sharing of partial applications in functional programs. In *Proc. Functional Programming Lang. and Computer Arch.,* pages 408–425. Springer Verlag, 1987.

[6] C. K. Gomard and P. Sestoft. Evaluation order analysis for lazy data structures. In Heldal, Hoist, and Wadler, editors, *Functional Programming, Glasgow Workshop 1991*, pages 112–127. Springer-Verlag, 1991.

[7] C. K. Gomard and P. Sestoft. Path analysis for lazy data structures. In M. Bruynooghe and M. Wirsing, editors, *Programming Language Implementation and Logic Programming, 4th International Symposium, PLILP '92, Leuven, Belgium. (Lecture Notes in Computer Science, vol. 631)*, pages 54–68. Springer-Verlag, 1992.

[8] Nevin Heintze. Set-based analysis of ML programs. In *Proc. ACM Conference on LISP and Functional Programming*, 1994.

[9] Fritz Henglein and Jesper Jørgensen. Formally optimal boxing. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 213–226, Portland, Oregon, January 1994.

[10] John Hughes. Compile-time analysis of functional programs. In David Turner, editor, *Research Topics in Functional Programming*, chapter 5. Addison-Wesley, 1990.

[11] John Hughes. Why functional programming matters. In David Turner, editor, *Research Topics in Functional Programming.* Addison-Wesley, 1990.

[12] John Launchbury. A natural semantics for lazy evaluation. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 144–154, Charleston, South Carolina, January 1993.

[13] Xavier Leroy. Polymorphism by name for references and continuations. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 220–231, Charleston, South Carolina, January 1993.

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

[15] A. Mycroft. *Abstract interpretation and optimizing transformations for applicative programs.* PhD thesis, Computer Science Dept,. Univ. of Edinburgh, 1981.

[16] Jens Palsberg. Closure analysis in constraint form. In *CAAP'94*, 1994.

[17] Simon L Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional language. In John Hughes, editor, *FPCA '91*, pages 636–666. Springer Verlag, 1991. LNCS 523.

[18] Peter Sestoft. *Analysis and efficient implementation of functional programs.* PhD thesis, DIKU, University of Copenhagen, Denmark, October 1991.

[19] Peter Sestoft. Deriving a lazy abstract machine. ID-TR 146, Dept. of Computer Science, Technical University of Denmark, September 1994.

[20] O. Shivers. The semantics of Scheme control-flow analysis. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, volume 26, pages 190–198, New Haven, CN, June 1991.

[21] G.S. Smith. Principal type schemes for functional programs with overloading and subtyping. *Science of Computer Programming*, 23: 197–226, December 1994.

[22] Yan-Mei Tang and Pierre Jouvelot. Separate abstract interpretation for control-flow analysis. In *TACS'94*, 1994.

[23] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 188–201, Portland, Oregon, January 1994.

[24] Philip Wadler. Deforestration: Transforming programs to eliminate trees. In Harald Ganzinger, editor, *ESOP '88*, pages 344–358. Springer Verlag, 1989.

[25] Mitchell Wand and Paul Steckler. Selective and lightweight closure conversion. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 435–445, Portland, Oregon, January 1994.

Type Systems for Closure Conversions

John Hannan
Department of Computer Science and Engineering
The Pennsylvania State University
University Park, PA 16802 USA

Abstract

We consider the problem of analyzing and proving correct simple closure conversion strategies for a higher-order functional language. We specify the conversions as deductive systems, making use of annotated types to provide constraints which guide the construction of the closures. We exploit the ability of deductive systems to specify concisely complex relationships between source terms and closure-converted terms. The resulting specifications and proofs are relatively clear and straightforward. The use of deductive systems is central to our work as we can subsequently encode these systems in the LF type theory and then code them in the Elf programming language. The correctness proofs can also be coded in this language, providing machine-checked versions of these proofs.

1 Introduction

Closure conversion is the process of transforming functions containing free variables into a closures, a representation of a function that consists of a piece of code for the function and a record containing the free variables occurring in the original function. This process consists not only of converting functions to closures but also of replacing function calls with the invocation of the code component of closures on the actual parameter and the closure itself (which will contain values for the free variables). Closure conversion is a critical step in the compilation of higher-order functional languages, and different closure conversion strategies can have remarkably different run-time behaviors in terms of space utilization. Reasoning about these conversions can become complicated as the conversion themselves become more complicated. We believe that a means for analyzing various conversion strategies will provide a useful tool for understanding and correctly implementing closure conversion.

1.1 Contribution

The main contribution of this paper is the development of type systems to specify and prove correct various closure conversion strategies. In particular, the type systems are reasonably simple and clearly express the relationship between source terms and closure converted terms. We specify the conversions as deductive systems axiomatizing judgments which relate expressions containing functions and those containing closures. These systems make critical use of annotated types to provide constraints which guide the construction of the closures. The use of deductive systems is critical to this work, as we subsequently encode these systems into the LF [7] type theory and then the Elf programming language [11], providing both experimental implementations of closure conversion but also machine-checked proofs of correctness. In the current paper we focus only on the deductive systems, but most of the systems presented here have been implemented and proved correct in Elf. We include only the deductive systems and statements of the relevant correctness theorems. For the full proofs and the Elf code implementing the closure conversions and specifying the proofs see the full version of the paper, available as a technical report from our institution.

The kinds of closure conversions addressed in this paper are simple, but the methods developed demonstrate the capabilities of type systems for describing closure conversions. Recent work on space efficient closure representation has demonstrated the efficiency possible if closures are carefully constructed using a variety of information [12]. While we have not yet considered such advanced closure conversion representations, we believe that our approach will provide a useful tool for reasoning about and proving correct such techniques.

1.2 Related Work

The problem of correctness for closure conversion has recently been addressed in [14]. The approach used in that work includes a flow analysis to generate constraints which ensures that the closure conversion algorithm generates closures that consistently use the correct procedure calling protocol in the presence of multiple calling protocols (for example, one protocol for use with closures as procedures and one for use with λ-abstractions as procedures). Based on techniques from abstract interpretation, their approach requires the introduction of an abstract notion of terms and evaluation (their "occurrence evaluator") and the relationship between their original language and this abstract version. Annotations are then added to provide constraint information and they prove that their conversion satisfies these constraints. Their proof of correctness, however, only shows what we called soundness in [6]: if the source program evaluates, then the converted term does too. (Wand proved the converse using different techniques in [13].) Our initial motivation was to demonstrate how equivalent results could be produced using type systems.

The idea of using type systems to specify constraints of programs and to guide the translation of programs has been successfully used by Tofte and Talpin to describe region inference for Standard ML programs. Region inference detects blocks of storage that can be allocated and deallocated in a stack-like fashion. Their use of annotated types has motivated some of our techniques for annotating function types with information regarding the free variables required to call the function.

Related work on compiler correctness includes [3] where compiler optimizations based on strictness analysis are proved correct. This work, however, considers CPS translations and definitions that resemble denotational semantics.

1.3 Organization of Paper

The remainder of the paper is organized as follows. In Sec. 2 we introduce a basic closure conversion specification and a verification of its correctness. In Sec. 3 we extend the basic conversion to a selective one and demonstrate how its correctness is a direct generalization of the basic case. In Sec. 4 we extend the selective conversion to one in which not all free variables need be included in a closure. Finally in Sec. 5 we conclude by mentioning some additional conversion strategies and our intent to verify them. In the appendix we give a brief introduction to the LF type theory and its application to specifying deductive systems.

2 Simple Closure Conversion

We begin by considering a simple closure conversion specification in which every function is converted into a closure.

2.1 Source and Target Languages

We consider just the simply typed λ-calculus as the source language:

$$E ::= x \mid \lambda x.E \mid E \mathbin{@} E$$

in which $E_1 \mathbin{@} E_2$ represents application. For our first presentation of closure conversion types play no role, and so we can also consider this method as applying to an untyped language. But in subsequent sections we rely heavily on types and a typed language.

The target language of closures consists of the following:

$$\begin{array}{lcl} M & ::= & x \mid n\#C \mid C \mid M \mathbin{@_c} M \\ C & ::= & c \mid [\lambda c.\lambda x.M, L] \\ L & ::= & \cdot \mid L, M \end{array}$$

in which $E_1 \mathbin{@_c} E_2$ represents application in which the value of E_1 will be a closure. The meta-variables M, C and L range over terms in the target language, closures (and closure variables), and lists of target terms, respectively. The λ-abstraction of the source language is replaced by the closure construction $[\lambda c.\lambda y.M, L]$ in which the bound variable c corresponds to the closure itself, the bound variable y corresponds to the bound variable of the λ-abstraction, M corresponds to the body of the λ-abstraction and L is a list of the free variables of the λ-abstraction. (We refer to c as the *closure-bound variable* of the closure.) We include the bound variable c to approximate the structure of closures as described in [2] in which a closure is invoked by fetching the first field of the closure and applying it to its arguments including the closure itself.

We represent variables in two ways: locally bound variables (i.e., variables bound by the nearest enclosing lambda abstraction in a source term) are represented as in the source language; non-locally bound variables are represented by the term $n\#C$ in which n is a positive integer (de Bruijn index) and C is either a closure or a closure-bound variable. For example, the term $\lambda x.\lambda y.(x\ y)$ is represented in the target language by the term

$$[\lambda c_1.\lambda x.([\lambda c_2.\lambda y.((1\#c_2)y), (\cdot, x)]), \cdot]$$

For both the source and target languages we can provide an operational semantics, each given by a set of inference rules which can be directly represented as LF signatures. Both semantics implement call-by-value to weak-head normal form. For the source language we introduce the judgment $e \hookrightarrow_s v$ and axiomatize it via the following two rules:

$$\frac{}{\lambda x.E \hookrightarrow_s \lambda x.E}$$

$$\frac{E_1 \hookrightarrow_s \lambda x.E' \qquad E_2 \hookrightarrow_s V_2 \qquad E'[V_2/x] \hookrightarrow V}{E_1\ @\ E_2 \hookrightarrow_s V}$$

For the target language we introduce the judgment $e \hookrightarrow_t v$ and axiomatize it via the following rules:

$$\frac{}{[\lambda c.\lambda y.M,\ L] \hookrightarrow_t [\lambda c.\lambda y.M,\ L]}$$

$$\frac{M_1 \hookrightarrow_t [\lambda c.\lambda y.M',\ L] \quad M_2 \hookrightarrow_t V_2' \quad M'[V_2'/y][[\lambda c.\lambda y.M',\ L]/c] \hookrightarrow_t V'}{M_1\ @_c\ M_2 \hookrightarrow_t V'}$$

$$\frac{nth\ N\ L\ V'}{N\#[\lambda c.\lambda y.M,\ L] \hookrightarrow_t V'}$$

The judgment ($nth\ N\ L\ V$) expresses the relation that the N^{th} element in the list L is V.

This specification of evaluation using closures focuses on the access of free variables in function bodies, but not on the mechanism by which values are stored into the record component of the closure. In the specification above, the substitution $M'[V_2/y]$ replaces all occurrences of y in M' with value V_2. As y may occur in the record component of a closure, this substitution can have the effect of loading the value V into any number of closures. While this is hardly realistic, this convention makes the specification particularly simple and easy to analyze. Our future goal is to provide further refinements of this specification to reflect more accurately the manipulation of closures.

2.2 The Closure Conversion Specification

To specify closure conversion we could have introduced a type system for source terms in which the type of a function explicitly provides information about the shape of the desired closure for the function. Then the translation from source to target languages would be trivial. We can, in fact, combine these two operations (typing and translating) into a single deductive system, in which types play a reduced role due to the presence of contextual information.

We specify closure conversion as a translation from source to target languages. We introduce the judgment $E \Longrightarrow M$ which denotes the property that source term E closure converts to term M. As we need some additional information to specify the conversion, we also introduce the judgment $\langle L, x, c\rangle \rhd E \Longrightarrow M; L'$ in which L and L' are lists of target language terms (typically variables), x is a source language term (typically a variable), c is a closure (or closure variable), E is a source term and M is a target term. The judgment can be read as follows: L is the list of variables (*or terms substituted for these variables*) that occur free and must therefore be included in an enclosing closure; x is the bound variable of the nearest enclosing λ-abstraction; c is the closure variable of the nearest enclosing closure (to be constructed); E is the source term to be converted; M is the converted term; and L' is the (possible) extension of L which includes the free variables of M. Note that the first judgment $E \Longrightarrow M$ is really just a special case of the second judgment where the sets L and L' are empty. We prefer to use two distinct judgments as it simplifies and clarifies the correctness proofs. Note also that while x and y in rules (1.2) and (1.4) denote variables in the source and target languages, respectively, in the remaining rules occurrences of x, y, z and c, though suggestive of variables, can range over arbitrary terms of the appropriate syntactic class (source term, target term or closure).

Finally, we need a third judgment, $L \triangleright x \mapsto N; L'$ which is used to generate a de Bruijn index N for a variable x. This judgment relies on the property that bound variables are distinct in the given source term. The judgment can be understood as follows: starting with (target) variable list L, source variable x closure converts to the N^{th} variable of L'. The list L' is different from L if only if the variable y to which x converts is not already in L. In this case, y is added to the end of L, creating L'. The complete system for basic closure conversion is given in Fig. 1.

The first two rules specify the top-level conversion. In rule (1.2) the universal quantification of the variables x, y, and c ensures that these variables are arbitrary (and do not already occur in any assumptions). (In our implementation in Elf this property is automatically maintained from our use of higher-order syntax and Π-quantification of these variables.) Note that the variable x may occur free in E and the variables y and c may occur free in M. The variables are bound by the universal quantifiers in the antecedent of the rule and are bound by λ-abstractions in the conclusion. This manipulation of variables, motivated by the higher-order syntax used in our implementation, eliminates any need for variable conventions or renaming. We use implication to introduce the hypothesis $x \Longrightarrow y$ (an instance of the judgment $E \Longrightarrow M$), instead of maintaining an explicit context of information. Again, our implementation supports this operation and structuring the rule in this way simplifies the correctness proofs. The operation $L + L'$ denotes the new list of variables obtained by appending to L those elements of L' (in order) not already occurring in L.

The structure of these rules, in particular, the use of universal quantification and implication is critical when we encode these rules into an LF signature and Elf program and apply the Propositions-as-Types analogy. Then, for example, the judgment $\forall x \forall y \forall c (x \Longrightarrow y \supset \langle \cdot, x, c \rangle \triangleright E \Longrightarrow M;\ L)$, when viewed as a type denotes a functional type taking four arguments (terms for x, y, and c, and an object of type $x \Longrightarrow y$). Thus a deduction of this judgment, when viewed as an object, represents a function which when applied to terms

$$\frac{E_1 \Longrightarrow M_1 \qquad E_2 \Longrightarrow M_2}{(E_1 \mathbin{@} E_2) \Longrightarrow (M_1 \mathbin{@_c} M_2)} \tag{1.1}$$

$$\frac{\forall x \forall y \forall c (x \Longrightarrow y \supset \langle \cdot, x, c \rangle \triangleright E \Longrightarrow M; L)}{\lambda x.E \Longrightarrow [\lambda c.\lambda y.M, L]} \tag{1.2}$$

$$\frac{\langle L, x, c \rangle \triangleright E_1 \Longrightarrow M_1; L' \qquad \langle L', x, c \rangle \triangleright E_2 \Longrightarrow M_2; L''}{\langle L, x, c \rangle \triangleright (E_1 \mathbin{@} E_2) \Longrightarrow (M_1 \mathbin{@_c} M_2); L''} \tag{1.3}$$

$$\frac{\forall x \forall y \forall c (x \Longrightarrow y \supset \langle \cdot, x, c \rangle \triangleright E \Longrightarrow M; L')}{\langle L, x', c' \rangle \triangleright \lambda x.E \Longrightarrow [\lambda c.\lambda y.M, L'] ; L + L'} \tag{1.4}$$

$$\frac{x \Longrightarrow y}{\langle L, x, c \rangle \triangleright x \Longrightarrow y; L} \tag{1.5}$$

$$\frac{L \triangleright z \mapsto N, L'}{\langle L, x, c \rangle \triangleright z \Longrightarrow (N \# c); L'} \; (z \neq x) \tag{1.6}$$

$$\frac{x \Longrightarrow y}{\cdot \triangleright x \mapsto 1; (\cdot, y)} \tag{1.7}$$

$$\frac{x \Longrightarrow y}{(L, y) \triangleright x \mapsto 1; (L, y)} \tag{1.8}$$

$$\frac{L \triangleright x \mapsto N; L'}{(L, y) \triangleright x \mapsto N{+}1; (L', y)} \tag{1.9}$$

Figure 1: Basic Closure Conversion

E', M', C' and an object representing the deduction $E' \Longrightarrow M'$, yields an object representing a deduction of $\langle \cdot, E', C' \rangle \triangleright E[E'/x] \Longrightarrow M[M'/y, C'/c]; L$.

2.3 Verifying the Conversion

We are now in a position to state the correctness criteria for our closure conversion specification. First we state two lemmas about the auxiliary judgments $L \triangleright x \mapsto N; L'$ and $\langle L, x, c \rangle \triangleright E \Longrightarrow M; L'$:

Lemma 1. *For all lists L, L', source term E, target term M and natural number N, if* $\vdash L \triangleright E \mapsto N; L'$ *and* $\vdash$ *(nth N L' M) then* $\vdash E \Longrightarrow M$.

The proof is by induction on the structure of the deductions for the judgment $L \triangleright E \mapsto N; L'$.

Lemma 2.

1. *For all source terms E, V, x, target term M, lists L, L', and closure term (or variable) C, if* $\vdash E \hookrightarrow_s V$ *and* $\vdash \langle L, x, C\rangle \triangleright E \Longrightarrow M; L'$ *then there exists a V' such that* $\vdash M \hookrightarrow_t V'$ *and* $\vdash V \Longrightarrow V'$;
2. *For all source terms E, x, target terms M, V', lists L, L', and closure term (or variable) C, if* $\vdash \langle L, x, C\rangle \triangleright E \Longrightarrow M; L'$ *and* $\vdash M \hookrightarrow_t V'$ *then there exists a V such that* $\vdash E \hookrightarrow_s V$ *and* $\vdash V \Longrightarrow V'$.

The proof is straightforward by induction on the structure of the deduction for $E \hookrightarrow_s V$ and then by cases on the structure of the deduction for $\langle L, x, c\rangle \triangleright E \Longrightarrow M; L'$. Finally, we can state the main theorem.

Theorem 3.

1. *For all source terms E, V and target term M, if* $\vdash E \hookrightarrow_s V$ *and* $\vdash E \Longrightarrow M$ *then there exists a V' such that* $\vdash M \hookrightarrow_t V'$ *and* $\vdash V \Longrightarrow V'$;
2. *For all source term E and target terms M, V', if* $\vdash E \Longrightarrow M$ *and* $\vdash M \hookrightarrow_t V'$ *then there exists a V such that* $\vdash E \hookrightarrow_s V$ *and* $\vdash V \Longrightarrow V'$.

This kind of correctness statement is in the same spirit as one of the correctness statements for the compiler found in our earlier work on compiler correctness [6].

3 Selective Closure Conversion

As pointed out in [14], avoiding closure creation plays an important role in generating efficient code for higher-order languages. In the basic closure conversion specification of the previous section, alI source language functions were converted to closures. We consider here the possibility of selective closure conversion in which source language functions are not converted to closures, but rather left as functions. Our focus in this section is not so much on determining exactly when closures need not be converted, but rather on demonstrating that both closures and functions can be handled together, ensuring that applications in the target language are constructed with the proper procedure calhng conventions.

The approach taken in [14] uses a relatively complex flow analysis, related to some abstract interpretation techniques. Using a type system, we provide a straightforward account of selective conversion. Instead of explicit constraints we have types, and type inference provides the means for resolving constraints imposed by these types. For demonstration purposes, we consider only one case in which a source language function is not translated into a closure: when the function contains no free variables.

3.1 Extending the Language

We begin by considering the types for our languages and by extending the target language. The types consist of some collection of base types and two kinds of function types:

$$\tau ::= a \mid \tau \to \tau \mid \tau \to_c \tau$$

A λ-abstraction will be given type $\tau_1 \to \tau_2$ if it should not be closure converted. A λ-abstraction will be given type $\tau_1 \to_c \tau_2$ if it should be closure converted.

The target language is extended by including λ-abstractions and a second form of application:

$$M ::= x \mid n\#c \mid [\lambda c.\lambda x.\ M,\ L] \mid M \ @_c\ M \mid \lambda x.M \mid M\ M$$

We extend the operational semantics for the language with the two rules:

$$\frac{}{\lambda y.M \hookrightarrow_t \lambda y.M}$$

$$\frac{M_1 \hookrightarrow_t \lambda y.M \qquad M_2 \hookrightarrow_t V_2 \qquad M[V_2/y] \hookrightarrow_t V}{M_1\ M_2 \hookrightarrow_t V}$$

3.2 Adding Selective Conversion

The closure conversion specification is modified in a few ways. First, we include the source language type in the judgments $E \Longrightarrow M$, $\langle L, x, c\rangle \triangleright E \Longrightarrow M; L'$, and $L \triangleright x \mapsto N; L'$. The judgments become $E \Longrightarrow M : \tau$, $\langle L, x, c\rangle \triangleright E \Longrightarrow M : \tau; L'$, and $L \triangleright x \mapsto N : \tau; L'$. Second, the original rules for translating λ-abstractions and applications use the function type $\tau_1 \rightarrow_c \tau_2$ to indicate that the λ-abstraction is converted to a closure and the application contains an operator that should evaluate to a closure. Finally, we add two new rules for when a λ-abstraction does not convert to a closure and the corresponding rules for the new application. The complete system is given in Fig. 2.

The rules for converting applications differ only in the type given to the operator. The new rules (2.3) and (2.7) for converting λ-abstractions differ from the original ones (in which closures are created) by requiring that the list of free variables occurring in the term be empty. This is enforced by the occurrence of '.' on the right-hand sides of the antecedents in these two new rules.

To ensure that this conversion produces terms that make proper use of closures we have the following consistency lemma.

Lemma 4. *For all source term E and target term M,*

1. *if $\vdash E \Longrightarrow M : \tau_1 \rightarrow \tau_2$ and $M \hookrightarrow_t V'$ then $V' = \lambda y.M'$ for some M';*
2. *if $\vdash E \Longrightarrow M : \tau_1 \rightarrow_c \tau_2$ and $M \hookrightarrow_t V'$ then $V' = [\lambda c \lambda y.M']$ for some M'.*

The proof is straightforward by induction of the structure of deductions. The fact that closed λ-abstractions need not be converted to closures is obvious and this lemma ensures that we can selectively convert λ-abstractions and still have correct procedure call protocols.

3.3 Verifying the Conversion

Adapting the proof of correctness for basic closure conversion to selective closure conversion is straightforward and the Elf program representing the

$$\frac{E_1 \Longrightarrow M_1 : (\tau_1 \rightarrow \tau_2) \qquad E_2 \Longrightarrow M_2 : \tau_1}{(E_1 \mathbin{@} E_2) \Longrightarrow (M_1 \mathbin{@} M_2) : \tau_2} \tag{2.1}$$

$$\frac{E_1 \Longrightarrow M_1 : (\tau_1 \rightarrow_c \tau_2) \qquad E_2 \Longrightarrow M_2 : \tau_1}{(E_1 \mathbin{@} E_2) \Longrightarrow (M_1 \mathbin{@_c} M_2) : \tau_2} \tag{2.2}$$

$$\frac{\forall x \forall y \forall c (x \Longrightarrow y : \tau_1 \ \supset \ \langle \cdot, x, c \rangle \ \triangleright E \Longrightarrow M : \tau_2 ; \cdot)}{\lambda x.E \Longrightarrow \lambda y.M : (\tau_1 \rightarrow \tau_2)} \tag{2.3}$$

$$\frac{\forall x \forall y \forall c (x \Longrightarrow y : \tau_1 \ \supset \ \langle \cdot, x, c \rangle \ \triangleright E \Longrightarrow M : \tau_2 ; L)}{\lambda x.E \Longrightarrow [\lambda c.\lambda y.M, L] : (\tau_1 \rightarrow_c \tau_2)} \tag{2.4}$$

$$\frac{\langle L, x, c \rangle \ \triangleright E_1 \Longrightarrow M_1 : (\tau_1 \rightarrow \tau_2); L' \quad \langle L', x, c \rangle \ \triangleright E_2 \Longrightarrow M_2 : \tau_1 ; L''}{\langle L, x, c \rangle \ \triangleright (E_1 \mathbin{@} E_2) \Longrightarrow (M_1 \mathbin{@} M_2) : \tau_2 ; L''} \tag{2.5}$$

$$\frac{\langle L, x, c \rangle \ \triangleright E_1 \Longrightarrow M_1 : (\tau_1 \rightarrow_c \tau_2); L' \quad \langle L', x, c \rangle \ \triangleright E_2 \Longrightarrow M_2 : \tau_1 ; L''}{\langle L, x, c \rangle \ \triangleright (E_1 \mathbin{@} E_2) \Longrightarrow (M_1 \mathbin{@_c} M_2) : \tau_2 ; L''} \tag{2.6}$$

$$\frac{\forall x \forall y \forall c (x \Longrightarrow y : \tau_1 \ \supset \ \langle \cdot, x, c \rangle \ \triangleright E \Longrightarrow M : \tau_1 ; \cdot)}{\langle L, x', c' \rangle \ \triangleright \lambda x.E \Longrightarrow \lambda y.M : (\tau_1 \rightarrow \tau_2); L} \tag{2.7}$$

$$\frac{\forall x \forall y \forall c (x \Longrightarrow y : \tau_1 \ \supset \ \langle \cdot, x, c \rangle \ \triangleright E \Longrightarrow M : \tau_1 ; L')}{\langle L, x', c' \rangle \ \triangleright \lambda x.E \Longrightarrow [\lambda c.\lambda y.M, L'] : (\tau_1 \rightarrow_c \tau_2);\ L + L'} \tag{2.8}$$

$$\frac{x \Longrightarrow y : \tau}{\langle L, x, c \rangle \ \triangleright x \Longrightarrow y : \tau ; L} \tag{2.9}$$

$$\frac{L, z \mapsto N : \tau, L'}{\langle L, x, c \rangle \ \triangleright z \Longrightarrow (N \# c) : \tau ; L'} \ (z \neq x) \tag{2.10}$$

$$\frac{x \Longrightarrow y : \tau}{\cdot \ \triangleright x \mapsto 1 : \tau ; (\cdot, y)} \tag{2.11}$$

$$\frac{x \Longrightarrow y : \tau}{(L, y) \ \triangleright x \mapsto 1 : \tau ; (L, y)} \tag{2.12}$$

$$\frac{L \ \triangleright x \mapsto N : \tau ; L'}{(L, y) \ \triangleright x \mapsto (N{+}1) : \tau ; (L', y)} \tag{2.13}$$

Figure 2: Selective Closure Conversion

proof is only slightly longer than the proof for basic conversion.

Theorem 5.

1. *For all source terms E, V und target term M, if* $\vdash E \hookrightarrow V$ *and* $\vdash E \Longrightarrow M \ : \ \tau$ *then there exists a V' such that* $\vdash M \hookrightarrow_t V'$ *and* $\vdash V \Longrightarrow V' \ : \ \tau$;
2. *For all source term E and target terms M, V', if* $\vdash E \Longrightarrow M \ : \ \tau$ *and* $\vdash M \hookrightarrow_t V'$ *then there exists a V such that* $\vdash E \hookrightarrow_s V$ *and* $\vdash V \Longrightarrow V' \ : \ \tau$.

4 Lightweight Closure Conversion

The point of a closure is to provide a function body with access to non-local variables at the time the function is called. In particular the function call which originally bound some of these variables may have returned by the time this closure is accessed. If, however, certain variables can be shown only to be accessed during the evaluation of the function body which bound them, then these variables do not necessarily need to be included as part of a closure, as their bindings will be available elsewhere. We can exploit this idea and reduce the number of variables included in a closure, possibly eliminating the need for a closure entirely in some cases. Detecting when this is possible requires detailed analysis of the expression being closure converted.

In related work we have developed a static escape analysis for λ-terms [5]. This analysis, presented as a type system, determines when a bound variable can escape its scope at run time, i.e., when the variable may be accessed even after the function in which it was bound has returned. This situation occurs when bound variables occur inside of function bodies and these functions can be returned as values of the function for which the variable is a formal parameter. An example illustrates this relatively simple idea. Consider the function $\lambda x.\lambda f.\ f\ x$. If this function is applied to some value v, then the result of the function call will be the closure consisting of the function $\lambda f.\ f\ x$ and the binding $x \mapsto v$. In this case, x is the bound variable of a function, but this variable continues to exist after returning from the function.

A variable simply occurring in the body of another function is not a sufficient condition to imply that it escapes. Consider the term $\lambda x.((\lambda f.f\ x)\ (\lambda y.y))$. By some simple observations of this term, we can see that the occurrence of x in the body will not escape its scope.

Our analysis uses a judgment of the form $\Gamma \rhd E\ :\ \tau \Rightarrow E^s$ in which Γ is a type context, E is a source term, τ is an annotated type, and E^s is an annotated term. The annotated target language is a typed λ-calculus, but it includes two forms for each construct in the source language: one regular form and one annotated form.

$$M ::= x \mid x^s \mid \lambda x.M \mid \lambda^s x^s.M \mid M\ @\ M \mid M\ @^s\ M$$

A term of the form $\lambda^s x^s.M$ indicates that the variable x^s cannot escape from its scope. The term $M\ @^s\ N$ indicates that the value of the term M will be a function whose bound variable cannot escape its scope.

The analysis essentially determines which lambda abstractions and applications in the source term can be annotated in the target term. For example, the two term given above could be annotated as

$$\lambda x.\lambda f^s.f^s\ x \quad \text{and} \quad \lambda x^s.((\lambda f^s.f^s\ x^s)(\lambda y^s.y^s)).$$

The annotations in the first term indicate that f^s cannot escape its scope but that x may. The annotations in the second term indicate that no variables can escape their scope. In [5] we use this information to provide a stack-based implementation of a functional language by first translating it into an annotated form. Annotated variables are allocated space on a run-time stack and can be deallocated when a function returns. Closures are only created at run time.

Applying this analysis to closure conversion we can observe that the only variables required for a function closure are those variables that occur free in the function *and* can escape their scope. If they can escape their scope, then their binding will not necessarily be part of the "global" environment at the time the function body is evaluated. Variables that cannot escape (those annotated after analysis) can be allocated on a stack (because they can be safely deallocated upon returning from the corresponding function call) and their values can always be accessed, when required, from this stack. Closures which do not contain such non-escaping variables are called *lightweight closures* in [14], though they do not focus directly on whether variables escape their scope. They refer to these non-escaping variables as *dynamic variables*.

We can specify such lightweight closure conversion as the composition of our escape analysis and an extended version of the conversion specification given in the previous section. Note that we could also combine these two into a single specification, but for clarity we will keep the two distinct and focus only on the extended version of closure conversion using annotated terms. The specification for lightweight conversion includes the previous rules for selective conversion (Fig. 2) and the additional rules of Fig. 3. The first five rules simply treat annotated λ-abstractions, applications and local variables as before in selective conversion. Rule (3.6) provides the essential difference. In this case, non-local, annotated variables are not included in the set of variables used for constructing closures. The variables are simply translated into the appropriate target language variables. Compare this rule with (2.10).

$$\frac{E_1 \Longrightarrow M_1 : (\tau_1 \rightarrow_c \tau_2) \qquad E_2 \Longrightarrow M_2 : \tau_1}{(E_1 \mathbin{@^s} E_2) \Longrightarrow (M_1 \mathbin{@_c} M_2) : \tau_2} \tag{3.1}$$

$$\frac{\forall x^s \forall y \forall c (x^s \Longrightarrow y : \tau_1 \supset \langle \cdot, x^s, c\rangle \rhd E \Longrightarrow M : \tau_2; L')}{\lambda^s x^s . E \Longrightarrow [\lambda c . \lambda y . M, L'] : (\tau_1 \rightarrow_c \tau_2)} \tag{3.2}$$

$$\frac{\langle L, x, c\rangle \rhd E_1 \Longrightarrow M_1 : (\tau_1 \rightarrow_c \tau_2); L' \quad \langle L', x, c\rangle \rhd E_2 \Longrightarrow M_2 : \tau_1; L''}{\langle L, x, c\rangle \rhd (E_1 \mathbin{@^s} E_2) \Longrightarrow (M_1 \mathbin{@_c} M_2) : \tau_2; L''} \tag{3.3}$$

$$\frac{\forall x^s \forall y \forall c (x^s \Longrightarrow y : \tau_1 \supset \langle \cdot, x^s, c\rangle \rhd E \Longrightarrow M : \tau_1; L')}{\langle L, x', c'\rangle \rhd \lambda^s x^s . E \Longrightarrow [\lambda c . \lambda y . M, L'] : (\tau_1 \rightarrow_c \tau_2);\ L + L'} \tag{3.4}$$

$$\frac{x^s \Longrightarrow y : \tau}{\langle L, x^s, c\rangle \rhd x^s \Longrightarrow y : \tau; L} \tag{3.5}$$

$$\frac{z^s \Longrightarrow y : \tau}{\langle L, x, c\rangle \rhd z^s \Longrightarrow y : \tau; L}\ (z \neq x) \tag{3.6}$$

Figure 3: Lightweight Closure Conversion

When using lightweight closures we do not explicitly pass dynamic variables to functions which need them (as done in [14]). Instead we expect an implementation to exploit the availability of the variables (actually, the values bound to them), located in some global store such as a stack. Some simple calculations allows each function to determine the location at run time of these dynamic variables on the stack [5].

To adequately characterize the correctness of lightweight closure conversion we would need to introduce an operational semantics for our closure language that provides a finer specification of storage than given by the one in Sec. 2. We leave this for future work.

5 Conclusions and Future Work

We have presented a series of closure conversion specifications using type systems. The systems are relatively simple and for basic and selective conversion we have constructed a machine-checked proof of correctness in Elf. A corresponding proof for lightweight conversion is in progress.

We are applying our technique to other strategies for closure conversion, with the expectation of constructing clear spec3cations for these strategies and also correctness proofs. Our goal is to model the space-efficient closure representations constructed in the Standard ML of New Jersey compiler [12]. An important aspect here is to represent closures in a manner which allows storage to be deallocated or reclaimed as soon as data is no longer needed. Before attempting these more complex closure representations and conversion strategies we need a solid understanding of some of the basic issues and techniques of closure conversion, and a suitable framework for expressing and reasoning about them. The current work provides such initial experience.

A common program transformation, prior to closure conversion is CPS translation which produces λ-terms which closely reflect the control-flow and data-flow operations of a traditional machine architecture. We have specified and proved correct the translation from source program to CPS program using the same approach given here, using deductive systems to specify operational semantics and the CPS translation. The proof is straightforward and captures the essential notion of continuations. We have not, however, combined this translation with closure conversion. This is the subject of future work.

We intend to analyze more detailed translation strategies that incorporate *caller-save* registers and *callee-save* registers as described in [2, 12]. Doing this will require more complex analyses but we expect that using type systems we can adequately express them. We intend to analyze and prove correct the notion of *safe for space* complexity as described in [2]. To accomplish this we will consider a variety of static analyses on programs including lifetime analysis and closure strategy analysis (which determines where to allocate each closure).

References

[1] Andrew Appel and Trevor Jim. Continuation-passing, closure-passing style. In *Conf. Rec. of the 16th ACM Symposium on Principles of Programming Languages,* pages 293–302, 1989.

[2] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[3] Geoffrey Burn and Daniel Le Métayer. Proving the correctness of compiler optimisations based on strictness analysis. In Maurice Bruynooghe and Jaan Penjam, editors, *Programming Languages Implementation and Logic Programming,* volume 714 of *Lecture Notes in Computer Science,* pages 346–364. Springer-Verlag, 1993.

[4] T. Coquand. An algorithm for testing conversion in type theory. In G. Huet and G. Plotkin, editors, *Logical Frameworks,* pages 255–279. Cambridge University Press, 1991.

[5] John Hannan. A type-based analysis for stack allocation in functional languages. Submitted, May 1995.

[6] John Hannan and Frank Pfenning. Compiler verification in LF. In Andre Scedrov, editor, *Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science,* pages 407–418. IEEE Computer Society Press, 1992.

[7] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM,* 40(1):143–184, 1993. A preliminary version appeared in *Symposium on Logic in Computer Science,* pages 194–204, June 1987.

[8] S. Michaylov and F. Pfenning. Natural semantics and some of its metatheory in Elf. In Lars Hallnäs, editor, *Extensions of Logic Programming,* pages 299–344 Springer-Verlag LNCS 596, 1992. A preliminary version

is available as Technical Report MPI-I-91-211, Max-Planck-Institute for Computer Science, Saarbrücken, Germany, August 1991.

[9] Frank Pfenning. Elf: A language for logic defition and verified metaprogramming. In *Fourth Annual Sympobum on Logic in Computer Science,* pages 313–322. IEEE Computer Society Press, June 1989.

[10] Frank Pfenning. An implementation of the Elf core language in Standard ML. Available via ftp over the Internet, September 1991. Send mail to elf-request@cs.cmu.edu for further information.

[11] Frank Pfenning. Logic programming in the LF logical framework. In G. Huet and G. Plotkin, editors, *Logical Frameworks,* pages 149–181. Cambridge University Press, 1991.

[12] Zhong Shao and Andrew W. Appel. Space-efficient closure representations. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Progmmming,* pages 150-161. ACM, ACM Press, 1994.

[13] Mitchell Wand. Correctness of procedure representations in higher-order assembly language. In *Proceedings of the Mathematical Foundations of Programming Semantics '91,* volume 598 of *Lecture Notes in Computer Science,* pages 294–311. Springer-Verlag, 1992.

[14] Mitchell Wand and Paul Steckler. Selective and lightweight closure conversion. In *Conf. Rec. 21st ACM Symposium on Principles of Programming Languages,* 1994.

A A Overview of LF

We briefly review the LF logical framework [7] as realized in Elf [9, 10, 11]. A tutorial introduction to the Elf core language can be found in [8].

The LF calculus is a three-level calculus for *objects*, *families*, and *kinds*. Families are classified by kinds, and objects are classified by *types*, that is, families of kind Type.

$$\begin{array}{lcll} \textit{Kinds} & K & ::= & \text{Type} \mid \Pi x : A.\ K \\ \textit{Families} & A & ::= & a \mid \Pi x : A_1.\ A_2 \mid \lambda x : A_1.\ A_2 \mid A\ M \\ \textit{Objects} & M & ::= & c \mid x \mid \lambda x : A.\ M \mid M_1\ M_2 \end{array}$$

Family-level constants are denoted by a, object-level constants by c. We also use the customary abbreviation $A \to B$ and sometimes $B \leftarrow A$ for $\Pi\ x : A.\ B$ when x does not appear free in B. Similarly, $A \to K$ can stand for $\Pi\ x : A.\ K$ when x does not appear free in K. A *signature* declares the kinds of family-level constants a and types of object-level constants c.

The notion of definitional equality we consider here is based on $\beta\eta$-conversion. Type-checking remains decidable (see [4]) and it has the advantage over the original formulation with only β-conversion that every term has an equivalent canonical form.

The Elf programming language provides an operational semantics for LF. This semantics arises from a computational interpretation of types, similar in spirit to the way a computational interpretation of formulas in Horn logic gives rise to Pure Prolog. Due to space limitations, we must refer the reader to [8, 9, 11] for further material on the operational semantics of Elf.

Throughout this paper we have used only deductive systems to present solutions to problems. Each of these systems, however, has a direct encoding as an LF signature (a set of constant declarations), and hence, also an Elf programs In particular, an Elf program corresponding to a verification proof, when type-checked, provides a (mostly) machine-checked version of the proof. For lack of space we have not provided the LF signatures or Elf programs corresponding to the systems given in the paper, but the ability to construct these is a critical aspect of our work. The Elf language provides a powerful tool for experimenting with, and verifying, various analyses.

We give here only a brief description of how the deductive systems and proofs described in the paper can be encoded as LF signatures. From there, the encoding of signatures into Elf programs is a direct translation: each signature item becomes an Elf declaration.

We represent a programming language (that we wish to study) via an abstract syntax consisting of a set of object constants for constructing objects of a particular type. For example, we introduce a type *tm* of source programs and collection of object constants for building objects of type *tm*. We use a higher-order abstract syntax to represent functions and this simplifies the manipulation of programs with bound variables.

We represent judgments such as $e \hookrightarrow v$ via a type family *eval*: $tm \to tm \to type$. For given objects $e : tm$ and $v : tm$, $(eval\ e\ v)$ is a type.

We represent inference rules as object level constants for constructing objects of types such as $(eval\ e\ v)$. For example, an inference rule

$$\frac{A_1 \quad A_2 \quad A_3}{A_0}$$

would be represented as a constant $c : \Pi\ x_1 : B_1 \cdots \Pi\ x_n : B_n(A_1^* \rightarrow A_2^* \rightarrow A_3^* \rightarrow A_0^*)$ in which A_i^* is the representation ofjudgment A_i as a type and the $x_i : B_i$ are the free variables (implicitly universally quantified) of the inference rule. Using such constants we can construct objects, for example, of type $(eval\ e\ v)$, representing the deduction $e \hookrightarrow v$.

Finally, we represent inductive proofs (with the induction over the structure of deductions) as signatures in which each constant represents a case in the inductive proof. For example, to prove a statement of the form "judgment A is derivable iff judgment B is derivable" then we define a new judgment or type family, for example $thm : A \rightarrow B \rightarrow type$ to express the relationship between objects of type A and objects of type B. Base cases in the inductive proof translate to axioms (objects of base type) while inductive step cases translate to inference rules (objects of functional type). See [6] for examples of this technique.

Effective Flow Analysis for Avoiding Run-Time Checks

Suresh Jagannathan and Andrew Wright
NEC Research Institute, 4 Independence Way,
Princeton, NJ 08540
{suresh,wright}@research.nj.nec.com

Abstract

This paper describes a general purpose program analysis that computes global control-flow and data-flow information for higher-order, call-by-value programs. This information can be used to drive global program optimizations such as inlining and run-time check elimination, as well as optimizations like constant folding and loop iuvariant code motion that are typically based on special-purpose local analyses.

The analysis employs a novel approximation technique called *polymorphic splitting* that uses let-expressions as syntactic clues to gain precision. Polymorphic splitting borrows ideas from Hindley-Milner polymorphic type inference systems to create an analog to polymorphism for flow analysis.

Experimental results derived from an implementation of the analysis for Scheme indicate that the analysis is extremely precise and has reasonable cost. In particular, it eliminates sigticantly more run-time checks than simple flow analyses (*i.e.* 0CFA) or analyses based on type inference.

1 Introduction

Advanced programming languages such as Scheme [3] and ML [15] encourage a programming style that makes extensive use of data and procedural abstraction. Higher degrees of abstraction generally entail higher run-time overheads; hence, sophisticated compiler optimizations are essential if programs written in these languages are to compete with those written in lower-level languages such as C. Furthermore, because Scheme and ML procedures tend to be small, these compiler optimizations must be interprocedural if they are to be effective.

Interprocedural optimizations require interprocedural control- and data-flow information that expresses where procedures are called and where data values are passed. In its purest form, the flow analysis problem for these languages involves determining the set of procedures that can reach a given application point and the set of values that can reach a given argument position in an application. This information can be used to drive global program optimizations such as inlining and run-time check elimination, as well as optimizations like constant folding and loop invariant code motion that are typically based on special-purpose local analyses.

We present a flow analysis framework for a call-by-value, higher-order language. The framework is parameterized over different approximations of exact values to abstract values, and hence can be used to construct a spectrum of analyses with different cost and accuracy characteristics [12, 22]. In particular, we study a novel approximation technique called *polymorphic splitting* that uses let-expressions as syntactic clues to gain precision. Polymorphic splitting borrows ideas from Hindley-Milner polymorphic type inference systems to create an analog to polymorphism for flow analysis. Polymorphic splitting allows the analysis to avoid merging information between unrelated calls to a polymorphic function, yielding more precise flow information than would be otherwise possible.

To investigate its effectiveness, we have implemented the analysis for Scheme. The implementation handles all of the constructs and operators specified in the R^{4}RS report [3], including variable-arity procedures, data structures, first-class continuations, and assignment. We use the analysis to avoid unnecessary run-time type checks at primitive operations. Run-time type checks can be avoided at an application of a primitive operator if the types of the arguments can be proven to conform to the expected signature of the operator being applied. As our analysis yields sets of abstract values that approximate the types of the arguments, it is easy to determine when a run-time check can be avoided: each argument's abstract value or type must be a subset of the type expected.

Experimental results for realistic Scheme programs indicate that polymorphic splitting is extremely precise and has reasonable cost. The analysis eliminates significantly more run-time checks than comparable simple analyses (*e.g.* 0CFA [21]) or type-inference based techniques (*e.g.* soft typing [24]). While the computational cost of our analysis appears to be higher than that of type-inference based methods, analysis times are still within reason for including the analysis in an optimizing compiler. Furthermore, and perhaps surprisingly, our polymorphic splitting analysis is often faster than coarse analysis such as 0CFA. We discuss the reasons for this Section 5.

The remainder of the paper is organized as follows. The next section informally motivates polymorphic splitting, and Section 3 defines our framework in a formal manner. Section 3.1 discusses polymorphic splitting, Section 4 discusses the implementation, and Section 5 provides performance measurements. The final section places our results in the context of related work.

2 Motivation

In the absence of any compile-time analysis, Scheme implementations must ensure *at run-time* that primitive operations are applied to arguments of a sensible type. Even in ML, where a static type discipline prevents some primitives from being applied to incorrect values, many primitive operations (*e.g.* hd) require run-time checks. These run-time checks can have a signficant negative impact on program performance.

Although one can construct *ad hoc* heuristics to determine when certain run-time checks can be safely removed, a more satisfactory solution is to use a general interprocedural analysis. Many analyses with different cost and accuracy characteristics are possible, and the right combination is not readily apparent for a given optimization. For example, consider the following Scheme expression:

```
    (let ((f (lambda (x) x)))
      (f 1))
```

Simple control-flow analyses [22] or set-based analyses [9] determine that the application of `f` to `1` produces `1` as the result. These simple low-order polynomial-time analyses effectively determine all potential call sites for all procedures, but *merge* the values of arguments from all call sites. The *type* of a procedure's argument is therefore the union of the types of all values to which the argument is possibly bound. This set of *abstract values* is propagated among expressions in the procedure's body, yielding a similarly merged set for the procedure's result.

Because of the way information is approximated, these analyses fail to recognize that a run-time check is unnecessary on the addition operation in the following expression:

```
(let ((f (lambda (x) x)))
   (f 1))
   (f #t)))
```

The arguments (`1` and `#t`) to the two calls to `f` are merged by such analyses to yield {`1,#t`} as the abstract value for `x` in `f`'s body. Consequently, any application of `f` in this framework yields {`1,#t`} as its result. Run-time check optimizations based on these analyses are unable to eliminate the unnecessary run-time check at the addition operation.

A more sophisticated analysis [22] avoids merging information in syntactically distinct calls to the same procedure by treating the call site at which the procedure is applied as a disambiguation context; this analysis is referred to as 1CFA. In the above example, a 1CFA analysis deduces that `f` in the first call is applied to `1` and returns `1` as its result and, that in the second call, `f` is applied to `#t` and returns `#t` . With 1CFA, the run-time check at the addition operation can be eliminated.

1CFA is an instance of a general flow-analysis framework based on *call-string* approximations. Call-string based analyses use the N most recent dynamic call sites as a disambiguation context, for some small, fixed N. Because the point at which a procedure is defined may be far removed from its point of use, call-string based analyses are highly sensitive to program structure. It is easy to construct a simple extension to the above example for which 1CFA computes overly conservative information:

```
(let ((f (lambda (x) x)))
   (g (lambda (a b) (a b))))
   (+ (g f 1))
   (g f #t)))
```

In this example, there is an intervening call between the point where `f` is first referenced and the point where it is applied. 1CFA computes an abstract value set for `f`'s result that contains both `1` and `#t`. Correctly disambiguating the two calls to `(lambda (x) x)` made via the two calls to `g` requires a 2CFA analysis that preserves two levels of call history.

In contrast, polymorphic type inference algorithms [14, 24] correctly infer the proper type for `f` without requiring such tuning. Polymorphic type systems disambiguate different calls to a polymorphic procedure by effectively duplicating the procedure wherever it is referenced. This yields an analysis that is insensitive to the dynamic sequence of calls separating the construction of a value from its use. Consequently, analyses based on call-string abstractions are only marginally useful in capturing polymorphism.

We therefore consider an alternative abstraction called *polymorphic splitting* that uses let-expressions as syntactic clues to determine when abstract evaluation contexts should be disambiguated. Let f be a procedure bound by a let-expression, and let $f_1, \ldots, f_n$ be syntactically distinct occurrences of f within the let-expression's body. An analysis incorporating polymorphic splitting associates a distinct abstract closure with each of the f_i. Thus, different applications of f will construct different abstract bindings for the arguments. The abstract values of these bindings are not merged with abstract values associated with bindings of other applications of f in the let-body. It is well-known that quantification rules for type variables can be discarded in favor of a substitution rule on polymorphic variables [16]. Polymorphic splitting captures the essence of polymorphism by incorporating this observation into a flow-analysis framework.

Like polymorphic type inference, polymorphic splitting relies only on lexical structure and is independent of dynamic contexts. In the above example, there are four distinct occurrences of let-bound variables in the let-body. The analysis effectively substitutes the closure values of these variables at the application points, and thereby avoids merging bindings from the diRerent applications. A run-time check optimization based on this analysis will eliminate all run-time checks from this example.

Some technical machinery is required to implement the intuition of polymorphism as substitution without actually performing an inefficient code transformation. We discuss this issue in Section 3.1.

3 The Framework

We present a formal definition of our analysis framework for a functional core language. The definition is sufficiently parameterized that we can easily modify it to obtain a spectrum of analyses with different cost and accuracy characteristics.

We consider a simple call-by-value language with *labeled expressions* e^l of the form

$$\begin{aligned} e^l ::= {} & c^l \mid x^l \mid (\lambda x.e_1^{l_1})^l \mid (e_1^{l_1} e_2^{l_2})^l \\ & \mid x^l_{[l']} \mid (\texttt{let}\ x_{[l]} = e_1^{l_1}\ \texttt{in}\ e_2^{l_2})^l \\ & \mid (\texttt{if}\ e_1^{l_1}\ \texttt{then}\ e_2^{l_2}\ \texttt{else}\ e_3^{l_3})^l \end{aligned}$$

where $c \in \mathit{Const}$ are *constants*, $x \in \mathit{Var}$ are *variables*, and $l \in \mathit{Label}$ are *labels*. Constants include simple values like `0, 1, true,` and `false.` Free variables (FV) and bound variables (BV) are defined as usual [2], except that a subscripted variable $x^l_{[l']}$ must be bound by a let-expression with label l', and an unsubscripted variable x^l must be bound by a λ-expression. A *program* $e_0^{l_0}$ is an expression with no free variables. We assume that bound variables are appropriately re-named so that all bound variables in a program are distinct. We require every subexpression of a program to have a unique label, and occasionally omit labels from expressions to avoid clutter. The exact semantics for this language is an ordinary call-by-value semantics [20]. Recursive procedures can be constructed with the call-by-value Y combinator.

Our analysis yields sets of *abstract values.* Abstract values are defined as follows:

$$\begin{array}{rcl@{\quad}c@{\quad}l}
v & \in & \mathit{Value} & = & \mathit{Label} + \mathit{Closure} \\
\langle l, \rho, \kappa \rangle & \in & \mathit{Closure} & = & \mathit{Label} \times \mathit{Env} \times \mathit{Contour} \\
\rho & \in & \mathit{Env} & = & \mathit{Var} \to \mathit{Contour} \\
\kappa & \in & \mathit{Contour} & &
\end{array}$$

A single abstract value v is either a label l or a closure $\langle l, \rho, \kappa \rangle$. An abstract value $v = l$ identifies a particular occurrence of a constant c^l, and hence represents a single exact value. An abstract value $v = \langle l, \rho, \kappa \rangle$ identifies a *set* of functions created by evaluations of a λ-expression $(\lambda x.e)^l$. In a typical operational semantics, a closure is a pair consisting of a λ-expression and an environment mapping its free variables to values. A single λ-expression may thus produce many different closures. An abstract closure $v = \langle l, \rho, \kappa \rangle$ approximates a set of exact closures created from the same λ-expression.

Several exact closures are mapped to a single abstract closure by collapsing their environments. An abstract closure's environment maps variables to *contours* rather than values. Contours are finite strings of labels. We can make different choices for the set of contours, and the set of contours selected governs the precision of the analysis. In call-string based analyses, contour labels identify application sites. In our analysis, contour labels identify either let-expressions or uses of let-bound variables.

A *flow analysis* of a program $e_0^{l_0}$ is a function

$$F \;:\; \mathit{ProgramPoint} \;\to 2^{\mathit{Values}}$$

that maps *program points* to sets of abstract values. A program point is either a *Var* × *Contour* or *Label* × *Contour* pair. *Var* × *Contour* pairs represent bindings constructed by function applications. *Label* × *Contour* pairs represent the results of expressions. Informally, a program point associates an abstract program state with an identifier or expression.

F is a flow analysis of a program $e_0^{l_0}$ if $\mathcal{A}[\![e_0^{l_0}]\!]\rho_0\kappa_0$ holds, where $\mathcal{A}$ is the relation defined in Fig. 1 and implicitly parameterized by F, and ρ_0 and κ_0 are a specific initial environment and initial contour. There are many functions which are flow analyses of $e_0^{l_0}$. The *least flow analysis* is the least such function. Intuitively, $\mathcal{A}$ specifies constraints on a graph defined by F. Edges in the graph correspond to subset constraints, and nodes in the graph correspond to program points. The addition of new information is modeled in the framework in terms of satisfiability of these constraints. The following

$$
\begin{array}{rl}
\mathcal{A}[\![c^l]\!]\rho\kappa \Rightarrow & l \in F\langle l,\kappa\rangle \\[1ex]
\mathcal{A}[\![(\lambda x.e)^l]\!]\rho\kappa \Rightarrow & \langle l, \rho|_{FV(\lambda x.e)}, \kappa\rangle \in F\langle l,\kappa\rangle \\[1ex]
\mathcal{A}[\![x^l]\!]\rho\kappa \Rightarrow & F\langle x, \rho(x)\rangle \subseteq F\langle l,\kappa\rangle \\[1ex]
\mathcal{A}[\![(e_1^{l_1}\ e_2^{l_2})^l]\!]\rho\kappa \Rightarrow & \mathcal{A}[\![e_1^{l_1}]\!]\rho\kappa \text{ and whenever } F\langle l_1,\kappa\rangle \neq \emptyset \\
& \quad \mathcal{A}[\![e_2^{l_2}]\!]\rho\kappa \text{ and whenever } F\langle l_2,\kappa\rangle \neq \emptyset \\
& \qquad \mathcal{A}[\![e_b^{l_b}]\!]\rho'[x \mapsto \kappa']\kappa' \text{ and} \\
& \qquad F\langle l_2,\kappa\rangle \subseteq F\langle x,\kappa'\rangle \text{ and} \\
& \qquad F\langle l_b,\kappa'\rangle \subseteq F\langle l,\kappa\rangle \\
& \qquad \text{where } (\lambda x.e_b^{l_b})^{l'} \in e_0^{l_0} \\
& \quad \text{for all } \langle l',\rho',\kappa'\rangle \in F\langle l_1,\kappa\rangle \\[1ex]
\mathcal{A}[\![(\mathsf{let}\ x = e_1^{l_1}\ \mathsf{in}\ e_2^{l_2})^l]\!]\rho\kappa \Rightarrow & \mathcal{A}[\![e_1^{l_1}]\!]\rho\kappa' \text{ and whenever } F\langle l_1,\kappa'\rangle \neq \emptyset \\
& \quad F\langle l_1,\kappa'\rangle \subseteq F\langle x,\kappa\rangle \text{ and} \\
& \quad F\langle l_2,\kappa\rangle \subseteq F\langle l,\kappa\rangle \text{ and} \\
& \quad \mathcal{A}[\![e_2^{l_2}]\!]\rho[x \mapsto \kappa]\kappa \\
& \text{where } \kappa' = \kappa l \\[1ex]
\mathcal{A}[\![x_{[l_0]}^l]\!]\rho\kappa \Rightarrow & l' \in F\langle l,\kappa\rangle \\
& \quad \text{for all } l' \in F\langle x,\rho(x)\rangle \\
& \langle l',\rho',\kappa'[l_0/l]\rangle \in F\langle x,\rho(x)\rangle \\
& \quad \text{for all } \langle l',\rho',\kappa'\rangle \in F\langle x,\rho(x)\rangle \\[1ex]
\mathcal{A}[\![\mathsf{if}\ e_1^{l_1}\ \mathsf{then}\ e_2^{l_2}\ \mathsf{else}\ e_3^{l_3}]\!]\rho\kappa \Rightarrow & \mathcal{A}[\![e_1^{l_1}]\!]\rho\kappa \text{ and whenever } F\langle l_1,\kappa\rangle \neq \emptyset \\
& \mathcal{A}[\![e_2^{l_2}]\!]\rho\kappa \text{ and} \\
& \mathcal{A}[\![e_3^{l_3}]\!]\rho\kappa \text{ and} \\
& F\langle l_2,\kappa\rangle \subseteq F\langle l,\kappa\rangle \text{ if } \mathsf{true}^{l'} \in e_0^{l_0} \text{ and } l' \in F\langle l_1,\kappa\rangle \\
& F\langle l_3,\kappa\rangle \subseteq F\langle l,\kappa\rangle \text{ if } \mathsf{false}^{l'} \in e_0^{l_0} \text{ and } l' \in F\langle l_1,\kappa\rangle
\end{array}
$$

The notation $\rho[x \mapsto \kappa]$ means the functional update or extension of ρ at x to κ. The notation $\kappa[l/l']$ means the contour derived by replacing l with l' in κ.

Figure 1: Relation $\mathcal{A}$

theorem establishes the existence of a least function F.

Theorem (Least Flow Analysis). *Given envkonment ρ_0, contour κ_0 and closed expession $e_0^{l_0}$, there exists a unique minimal function* F *such that $\mathcal{A}[\![e_0^{l_0}]\!]\rho_0\kappa_0$ holds.*

Proof Sketch. Since the analysis only manipulates a finite set of values, we can enumerate the set S of functions F for which $\mathcal{A}$ holds, and impose a natural partial ordering on S. Suppose F_1 and F_2 are in S. By the structure of the constraint rules, we show that $\mathcal{A}$ must also hold for $F_1 \cap F_2$. ■

The application rule introduces two constraints. The first requires that the abstract value of the argument be a subset of the value of the formal; in other words, information flows from the actual parameter of the call to the function's formal. The second requires that the abstract value of the function body evaluated at this call site be a subset of the abstract value of the application; in other words, information flows from the function body to the call site. Besides these two constraints, the rule introduces an explicit strictness ordering. If the abstract value of the function position at a call is unspecified, i.e. no information flows into the node represented by the corresponding program point, the argument position need not be evaluated. A similar strictness constraint is introduced to cutoff evaluation of the function body if the abstract value of the argument position is unspecified. This strictness constraint corresponds to a reachability assertion [12], and significantly reduces the number of abstract values generated by the analysis.

3.1 Polymorphic Splitting

The last two rules of Fig. 1 for let-expressions and let-bound variables are the only rules that introduce new contours. For programs that do not use let-expressions, contours are essentially useless—all expressions are evaluated in the initial contour. Thus, in the absence of let, the analysis computes the same information as a 0CFA analysis. Polymorphic splitting uses different contours created at let-expressions and let-bound variables to disambiguate different uses of a let-bound procedure.

A let-binding evaluates in a new contour κ' which is constructed by appending the let-expression's label l to the let-expression's contour κ. The maximum length of a contour string is therefore the maximum nesting depth of let-bindings. Any closures created while evaluating the let-binding will capture the extended contour κ' as their third component (see the rule for λ-expressions). To illustrate, we adapt the second example from Section 2 to our formal core language:

$$(\texttt{let}\ f = (\lambda x.x^{l_1})^{l_2}\ \texttt{in}\ ((\lambda d.(f^{l_3}_{[l_0]}\ 1^{l_4})^{l_5})^{l_6}\ (f^{l_7}_{[l_0]}\ \#t^{l_8})^{l_9})^{l_{10}})^{l_0}$$

The let-binding evaluates in contour $\kappa_0 l_0$, and returns the abstract value set $\{\langle l_2, \emptyset, \kappa_0 l_0\rangle\}$. This abstract value set is bound to f in program point $\langle f, \kappa_0\rangle$.

When a let-bound variable $x^l_{l'}$, is used, abstract closures bound to the variable are split. For each abstract closure c that captured the let-expression label l', we define a new closure c' whose contour is derived by substituting l for l' in c's contour component. We return the new closure c' rather than c. In the example above, the abstract value set for f at $f^{l_3}_{[l_0]}$ is $\{\langle l_2, \emptyset, \kappa_0 l_0\rangle\}$. We split the closure in this set by substituting l_3 for l_0 in the closure's contour, yielding $\{\langle l_2, \emptyset, \kappa_0 l_3\rangle\}$ as the abstract value set for $f^{l_3}_{[l_0]}$. Similarly, $\{\langle l_2, \emptyset, \kappa_0 l_7\rangle\}$ is the abstract value set for $f^{l_7}_{[l_0]}$ When these closures are applied, their argument bindings are created in different contours ($\kappa_0 l_3$ and $\kappa_0 l_7$) and the closures evaluate in different contours. The analysis thereby avoids merging bindings from the two calls.

As another example, consider the expression

$$((\lambda\ f.((f\ \texttt{0})\ (f\ \texttt{true})))\ (\lambda x.(\lambda y.x)))$$

which evaluates to `0`, and whose least flow analysis is shown in Fig. 2. The set of abstract values for the result of this expression (program point $\langle l_0, \kappa_0\rangle$) is $\{l_2, l_5\}$. The analysis produces an imprecise result because the two applications $(f\ \texttt{0})^{l_3}$ and $(f\ \texttt{true})^{l_6}$ each create the same abstract closure. In an exact semantics, these two applications create different closures binding different values for their free variable x.

Rewriting the above example using a let-expression enables polymorphic splitting to take place:

$$(\texttt{let}\ f = \lambda x.\lambda y.x\ \texttt{in}\ ((f\ 0)\ (f\ \texttt{true}))$$

Fig. 3 presents the least flow graph of this rewritten expression. Several closures are now created for f with contours $\kappa_0 l_0, \kappa_0 l_1$, and $\kappa_0 l_4$. Since there

$$\begin{array}{ll}
\langle l_1, \kappa_0 l_0 \rangle \mapsto \{\langle l_{11}, \emptyset, \kappa_0 l_1 \rangle\} & \langle l_9, \kappa_0 l_1 \rangle \mapsto \{l_2\} \\
\langle l_2, \kappa_0 l_0 \rangle \mapsto \{l_2\} & \langle l_{10}, \kappa_0 l_1 \rangle \mapsto \{\langle l_{10}, [x \mapsto \kappa_0 l_1], \kappa_0 l_1 \rangle\} \\
\langle l_3, \kappa_0 l_0 \rangle \mapsto \{\langle l_{10}, [x \mapsto \kappa_0 l_1], \kappa_0 : l_1 \rangle\} & \langle l_{10}, \kappa_0 l_4 \rangle \mapsto \{\langle l_{10}, [x \mapsto \kappa_0 l_4], \kappa_0 l_4 \rangle\} \\
\langle l_4, \kappa_0 l_0 \rangle \mapsto \{\langle l_{11}, \emptyset, \kappa_0 l_4 \rangle\} & \langle l_{11}, \kappa_0 l_0 \rangle \mapsto \{\langle l_{11}, \emptyset, \kappa_0 l_0 \rangle\} \\
\langle l_5, \kappa_0 l_0 \rangle \mapsto \{l_5\} & \langle l_0, \kappa_0 l_0 \rangle \mapsto \{l_2\} \\
\langle l_6, \kappa_0 l_0 \rangle \mapsto \{\langle l_{10}, [x \mapsto \kappa_0 l_4], \kappa_0 l_4 \rangle\} & \langle f, [l_0] \rangle \mapsto \{\langle l_{11}, \emptyset, \kappa_0 l_0 \rangle\} \\
\langle l_7, \kappa_0 l_0 \rangle \mapsto \{l_2\} & \langle x, [l_1] \rangle \mapsto \{l_2\} \\
 & \langle x, [l_4] \rangle \mapsto \{l_5\} \\
 & \langle y, [l_1] \rangle \mapsto \{\langle l_{10}, [x \mapsto \kappa_0 l_4], \kappa_0 l_4 \rangle\}
\end{array}$$

Figure 2: The least flow analysis of
$e_0^{l_0} = ((\lambda f.((f^{l_1} 0^{l_2})^{l_3}\ (f^{l_4}\ \texttt{true}^{l_5})^{l_6})^{l_7})^{l_8}\ (\lambda x.(\lambda y.x^{l_9})^{l_{10}})^{l_{11}})^{l_0}$

are now two separate bindings for x corresponding to the two arguments passed to f, this analysis yields a more precise result.

3.2 Comparison to Type Inference

For several important idioms, polymorphic-splitting results in finer precision than Hindley-Milner typing or safety analysis [18] as embodied by a 0CFA analysis. As a simple illustration, consider the expression:[1]

$$\begin{array}{ll}
\langle l_1, \kappa_0 \rangle \mapsto \{\langle l_{11}, \emptyset, \kappa_0 \rangle\} & \langle l_9, \kappa_0 \rangle \mapsto \{l_2,\ l_5\} \\
\langle l_2, \kappa_0 \rangle \mapsto \{l_2\} & \langle l_{10}, \kappa_0 \rangle \mapsto \{\langle l_{10}, [x \mapsto \kappa_0], \kappa_0 \rangle\} \\
\langle l_3, \kappa_0 \rangle \mapsto \{\langle l_{10}, [x \mapsto \kappa_0], \kappa_0 \rangle\} & \langle l_{11}, \kappa_0 \rangle \mapsto \{\langle l_{11}, \emptyset, \kappa_0 \rangle\} \\
\langle l_4, \kappa_0 \rangle \mapsto \{\langle l_{11}, \emptyset, \kappa_0 \rangle\} & \langle l_0, \kappa_0 \rangle \mapsto \{l_2,\ l_5\} \\
\langle l_5, \kappa_0 \rangle \mapsto \{l_5\} & \langle f, \kappa_0 \rangle \mapsto \{\langle l_{11}, \emptyset, \kappa_0 \rangle\} \\
\langle l_6, \kappa_0 \rangle \mapsto \{\langle l_{10}, [x \mapsto \kappa_0], \kappa_0 \rangle\} & \langle x, \kappa_0 \rangle \mapsto \{l_2,\ l_5\} \\
\langle l_7, \kappa_0 \rangle \mapsto \{l_2,\ l_5\} & \langle y, \kappa_0 \rangle \mapsto \{\langle l_{10}, [x \mapsto \kappa_0], \kappa_0 \rangle\} \\
\langle l_8, \kappa_0 \rangle \mapsto \{\langle l_8, \emptyset, \kappa_0 \rangle\} &
\end{array}$$

Figure 3: The least flow analysis of
$e_0^{l_0} = (\texttt{let}\ f_{l_0} = (\lambda x.(\lambda y.x^{l_9})^{l_{10}})^{l_{11}}\ \texttt{in}\ ((f_{[l_0]}^{l_1} 0^{l_2})^{l_3} (f_{[l_0]}^{l_4}\ \texttt{true}^{l_5})^{l_6})^{l_7})^{l_0}$

$(\texttt{let}\ h = (\lambda x.\lambda y.$

[1]The examples in this section use natural extensions for begin and primitive operators.

```
      if $x^{l_1}$
      then false
      else $y^{l_2} + 1)^{l_3}$
  in begin
    $(1 + (h^{l_4}_{[l_0]}$ false $1)^{l_5})^{l_6}$
    $(h^{l_7}_{[l_0]}$ true ``foo''$)^{l_8}$
    end $)^{l_0}$
```

Under a standard Hindley-Milner type discipline, this program would fail to type-check for two reasons. First, the branches of the conditional have different types. Second, y is assumed to be of type `Int` based on the context in which it is used in the procedure body; the second call to h would violate this assumption. Under a 0CFA analysis, the two calls to h would be merged producing an abstract value for y that contains both 1 (from the first call) and `''foo''` (from the second). Depending on how the analysis is used, this merging would either lead to the program being rejected, or would result in run-time type-checks being required at both calls to h, and in the addition operation within h's body.

Both static type inference and 0CFA ignore inter-variable dependencies. Consequently, they are unable to capture the fact that y is an integer precisely when x is `false`. The flow analysis described here captures this information. In particular, the abstract values at all program points with label l_2 do not include the symbol `''foo''` since control never reaches the false branch of the conditional in the second call. Moreover, the abstract value at all program points with label l_8 will contain only `false`, thus allowing such calls to be used in a Boolean context even though the conditional yields an integer in the false branch. This kind of precision captures a form of polymorphism well-suited to Scheme programs, and has a direct impact on performance.

Although polymorphic splitting records inter-variable dependencies, the approximation defined by relation $\mathcal{A}$ does lose precision relative to Hindley-Milner typing in some cases. In particular, free occurrences of a polymorphic procedure defined within another may get merged because of the way contours are extended and substituted. Consider the following example:

```
(let (f = (λx.x^{l_f})
in  (let  g = λx.λy. ... (f^{l_5} x) ... (f^{l_6} y) ...
    in    (g^{l_3} ...)
          (g^{l_4} ...))^{l_1})^{l_0}
```

To ftithfully model polymorphism, we require f to be evaluated in four distinct contours. However, because g's closure binds f to contour $\kappa_0 l_0$, the first occurrence of f in g will be merged over the two calls to g, as will the second. For example, the abstract value yielded by evaluating the first reference to f in g will be a set containing the abstract closure $\langle l_f, \rho_0, [\kappa_0 l_5] \rangle$; evaluation of this occurrence of f in the second call to g will be the same.

It is possible to extend $\mathcal{A}$'s definition to disambiguate polymorphic references such that it fully captures the behavior of Hindley-Milner type inference. However, it is not clear whether such disambiguation in a flow-analysis context can be performed without making the implementation of the analysis too expensive for realistic programs. Our experimental results indicate that the potential loss of precision for programs that have nested polymorphic procedures does not compromise the practical utility of polymorphic splitting eliminating run-time checks.

4 Implementation

The implementation of the analysis and the run-time type-check optimization consists of approximately 4000 lines of Scheme code. The output of the analysis is used to control the placement of three kinds of run-time checks:

1. An *arity check* is required at a λ-expression if the procedure may be applied to an inappropriate number of arguments.
2. An *application check* is required at an application if the expression in the call position may yield a value other than a closure.
3. A *primitive check* is required at a primitive operation if the operation may be applied to a value of inappropriate type.

The analysis operates over the entire Scheme language, and thus supports variable-arity procedures, continuations, data structures, and assignment. While most extensions to the functional core are straightforward, there are several worthy of mention:

1. *Data Structures.* Elements of Scheme data structures can be mutated. The analysis tracks such assignments by recording them in the program point that holds the value of the corresponding sub-expression. Unlike other static type systems proposed for Scheme [24], assigning to data structures causes no loss in precision. Assigning to a field in a pair, for example, simply augments the abstract value set for that field.

2. *Implicit Allocation.* Certain Scheme constructs implicitly allocate storage. The most obvious example is variable-arity procedures, *i.e.*, procedures that take an optional number of arguments. Optional arguments are bundled into lists which are implicitly allocated and bound to a *rest* argument when the procedure is applied. Since expressions in the body of the procedure can walk down the rest argument using regular list operations, the implementation uses extra labels at each application to hold rest arguments.

3. *Type Predicates and Conditionals.* A common Scheme idiom is to use type predicates in conditional tests to guarantee that variables have a desired type in appropriate branches of the conditional. Failure to take such type predicates into account in the analysis can lead to unnecessary merging of abstract value sets and loss of precision. Our implementation recognizes type predicates that satisfy simple syntactic conditions, and evaluates the branches of a conditional in separate environments. These environments bind the variable upon which the type predicate test is performed to different abstract values consistent with the predicate check. (This environment splitting is *not* reflected in Fig. 1.)

5 Performance

Our implementation runs on top of Chez [5], a commercially available implementation of Scheme. At optimize-level 3, Chez eliminates almost all run-time checks, making no safety guarantees. By feeding the output of the analysis to a procedure that inserts explicit run-time checks based on the categories described in the previous section, the resulting program can be safely executed at optimize-level 3.

We have used the analysis to eliminate run-time checks from moderately sized Scheme programs. Fig. 4 lists the benchmarks used to test the analysis, their size in number of lines of code, the number of sites where run-time checks would ordinarily be required in the absence of any optimizations, and the time to analyze them under polymorphic splitting, soft typing [24], a 0CFA implementation, and a 1CFA implementation. The times were gathered on a 150 MHz MIPS R4400 with 1 GByte of memory.

Benchmark	**Lines**	**Sites**	**Analysis Time (in seconds)**			
			Polymorphic Splitting	**Soft Typing**	**0CFA**	**1CFA**
Lattice	215	252	.26	.46	.13	.49
Browse	233	283	.21	.96	.18	.50
Check	278	376	1.94	1.76	12.42	10.97
Graphs	621	413	.30	.73	.22	1.10
Boyer	632	212	2.36	3.69	35.70	80.83
N-Body	874	1184	1.06	2.75	.44	*
Dynamic	2331	2370	40.79	4.68	190.49	*
Nucleic	3334	911	66.43	6.44	61.75	64.91
Nucleic2	3264	1127	252.70	21.67	207.24	261.26

*** For N-Body and Dynamic, the 1CFA analysis exhausted heap space.**

Figure 4: Benchmark programs, their sige (in lines of code), static incidences of run-time checks for these programs in the absence of any run-time check optimization, and analysis times under polymorphic splitting, soft typing, 0CFA, and 1CFA.

The program `Lattice` enumerates the lattice of maps between two lattices, and is purely functional. `Browse` is a database searching program that allocates extensively. The program `Check` is a simple static type checker for a subset of Scheme. `Graphs` counts the number of directed graphs with a distinguished root and k vertices, each having out-degree at most 2. This program makes extensive use of mutation and vectors. `Boyer` is a term-rewriting theorem prover that allocates heavily. `N-Body` is a Scheme implementation [25] of the Greengard multipole algorithm [7] for computing gravitational forces on point-masses distributed uniformly in a cube. `Dynamic` is an implementation of a tagging optimization algorithm [10] for Scheme. `Nucleic` is a constraint satisfaction algorithm used to determine the three-dimensional structure of nucleic acids [6]. It is floating-point intensive and uses an object package implemented using macros and vectors. `Nucleic2` is a modified version of `Nucleic` described below.

For several of the benchmarks, the analysis time required by soft typing is less than the time required for either form of flow analysis. Because soft typing is based on a Hindley-Milner type inference framework [11, 14], the body of a λ-expression is evaluated only once. Applications unify the type signature for a procedure's arguments with the type inferred for the formal, and thus do not require re-analysis of the procedure body. In contrast, the implementation of the constraint rules specified for our flow analysis propagates the value of a procedure's argument to the sites where it is referenced within the procedure body. Thus, expressions within a procedure may be evaluated each time an application of the procedure is evaluated. Despite the potential for re-evaluation of expressions in a λ-body, the analysis times for polymorphic splitting are reasonably close to that of soft typing; `Dynamic` and `Nucleic` are the only exceptions.

In many cases, the analysis time for polymorphic splitting is less than the analysis time for 0CFA. On the surface, this is counter-intuitive: one might imagine a more precise analysis would have greater cost in analysis time. The reason for the significant difference in analysis times is because 0CFA yields coarser approximations, and thus induces more merging. More merging leads to more propagation, which in turn leads to more re-evaluation. Consider an abstract procedure value P. This value is represented as a set of abstract closures. As the size of this set becomes bigger, abstract applications of P require analyzing the body of more λ-expressions since the application rule applies the abstract value of the argument to *each* element in the set defined by P. Because polymorphic splitting results in less merging than 0CFA, abstract value sets are smaller, thus leading to shorter analysis times.

Fig. 5 shows the percentage of *static* checks remaining in these benchmarks under the different analyses after run-time check optimization has been applied. The static iounts measure the textual number of run-time checks remaining in the programs. Fig. 6 shows the percentage of *dynamic* checks remaining in the benchmarks. The dynamic counts measure the number of times these checks are evaluated in the actual execution of the program. Dynamic counts provide a reasonable metric to determine the effectiveness of an analysis—a good analysis will eliminate run-time checks along control-paths most likely to be exercised during program execution.

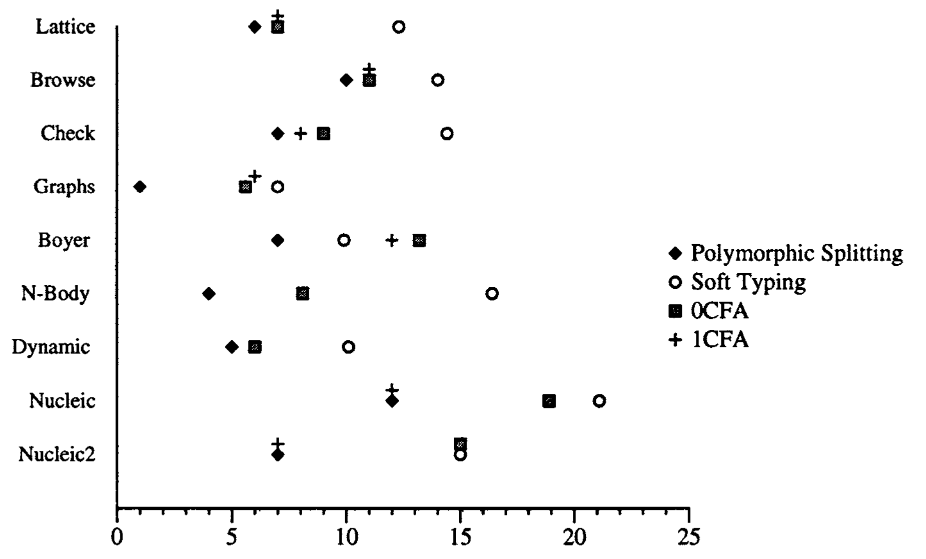

Figure 5: Percentage of static checks remaining after run-time check optimization.

For each of the benchmarks, polymorphic splitting requires fewer run-time checks than either 0CFA or soft typing. In many cases, the difference is significant. Interestingly, 0CFA also outperforms soft typing on several benchmarks. For example, the `Lattice` program has roughly half as many static checks when analyzed using 0CFA than using soft typing. The primary reason for this is the problem of *reverse flow* [24] in the soft-typing framework that leads to imprecise typing. Reverse flow refers to type information that flows both with and counter to the direction of value flow. Conditionals often cause reverse flow because the type rules for a conditional require both its branches to have the same type. Consequently, type information specific to one branch may cross over to the other, yielding a weaker type signature than necessary for the entire expression. Because of reverse flow, a cascading effect can easily occur in which many unnecessary run-time checks are required because of overly conservative type signatures.

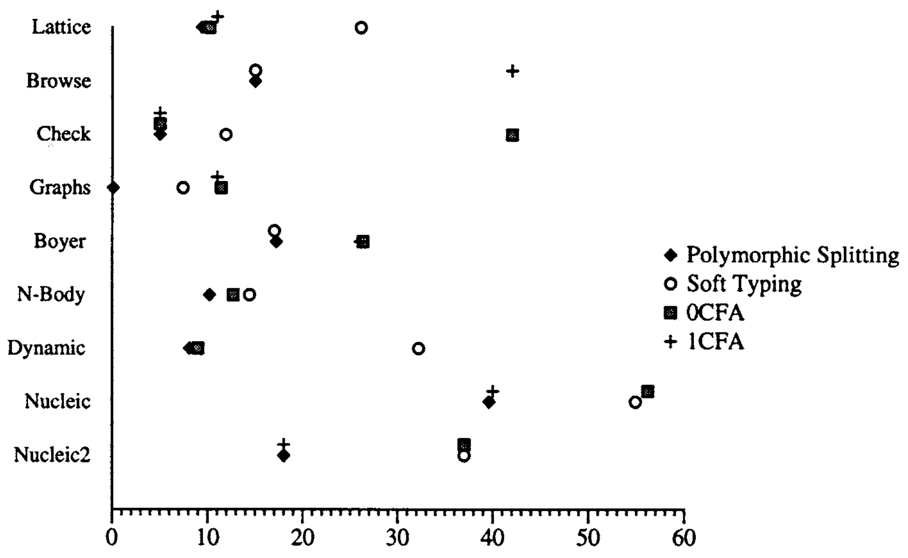

Figure 6: Percentage of dynamic checks remaining after run-time check optimbation.

The dynamic check count figures indicate that polymorphic splitting eliminates run-time checks at important program sites. The improvement over soft typing is more striking here. For example, about one third as many run-time checks are encountered during the execution of Lattice optimized under polymorphic splitting than under soft typing. About one quarter as many checks are executed for `Dynamic`. Here also, a simple 0CFA flow analysis generally does better than soft typing, although for some benchmarks such as `Boyer` and `Graphs`, 0CFA is worse. This indicates that the approximation used by polymorphic splitting is superior to 0CFA. Insofar as the benchmark programs use common Scheme idioms and programming styles, we conclude that Scheme programs use polymorphism in interesting and non-trivial ways. Analyses which are sensitive to polymorphism will outperform those that are not.

The counts of static and dynamic checks for `Nucleic` are significantly higher than for the other benchmarks. `Nucleic` implements a structure package that supports a simple form of inheritance; the main data objects manipulated by the program are defined in terms of structure instances. Structures are implemented as vectors, and tags (represented as symbols) are interspersed in this vector to demarcate distinct substructures in the inheritance hierarchy. Since the analyses do not retain distinct type information for individual vector slots, references to these structures invariably incur a run-time check. Thus, although the vast majority of references in the program are to floating-point data, all the analyses require run-time checks at arithmetic operations that are applied to elements of these structures. `Nucleic2` is a modified version of `Nucleic` which uses an alternative representation for structures. This representation separates the tags from the actual data stored in structure slots, enabling the analyses to avoid run-time checks when extracting elements from structures. `Nucleic2` incurs significantly fewer run-time checks than `Nucleic` for all analyses, although these improvements come at the cost of increased analysis times.

6 Conclusions and Related Work

A flow analysis can enable and facilitate a number of important optimizations necessary to implement realistic high-level languages efficiently. Run-time check elimination is one such example that is relevant in the context of languages such as Scheme or ML.

Although the flow analysis problem has been well-studied [13], and although parameterizable systems have been investigated elsewhere [12, 22], the applicability of such frameworks for optimizing realistic programs has enjoyed relatively little examination. When viewed in the context of run-time check elimination, flow analysis bears an interesting relationship to type inference [19]. However, there has been little work on extending the relation to polymorphism or to understand its implications on implementations.

One important kind of run-time check optimization is elimination of type tags [23]. Shivers [21] used the term *type recovery* to describe a type-tag elimination optimization based on flow analysis. Because this framework relies on call-string abstractions, and did not study the possibility of exploiting polymorphism, its success was limited. Moreover, since no attempt was made to implement the interpretation for the entire Scheme language, making a quantified assessment of its utility is difficult. Henglein [10] describes a tagging optimization based on type inference. While the analysis can reduce tagging overheads in many Scheme programs, it does not consider polymorphism or union types, and uses a coarse type approximation that is significantly more imprecise than control-flow analysis via polymorphic splitting.

Soft typing [24] and Infer [8] are two other type systems implemented for Scheme that employ traditional type inference techniques to derive type information which can be then used to eliminate or obviate run-time checks. Infer is a *statically* typed polymorphic dialect of Scheme. Like ML, certain well-defined programs (as determined by Scheme's dynamic semantics) will be prohibited under Infer because of apparent type errors it detects. Soft typing is a less restrictive alternative. Soft typing is a genertiation of static type checking that accommodates both dynamic typing and static typing in one framework. A soft type checker infers types for identifiers and inserts run-time checks to transform untypable programs to typable form. Aiken *et al.* [1] describe a more sophisticated soft type system for a functional language. This system uses conditional types in a more powerful type language that should yield more a precise analysis than the soft type system mentioned above, but no implementation is available for a realistic programming language.

Besides type inference and abstract interpretation, there has been recent work on using constraint systems [9, 17] to analyze high-level programs. These systems are based on an operational semantics that ignores all inter-variable dependencies. Consequently, while efficient implementations of these analyses can be built, it is unclear whether they provide the necessary precision to perform useful run-time check optimizations. Refinements on these approaches that take into account polymorphism are possible [9], but are *ad hoc* and do not fit neatly within the constraint framework.

We conclude that flow analysis offers the possibility of distilling more precise information useful for run-time check elimination than unification-based type inference procedures. Because flow analysis tracks dynamic control-flow, it can avoid incorporating information propagated from dead or unreachable code in abstract values. Furthermore, because these analyses record the creation points of values, they can be used to build a more refined notion of the set of values that can be associated with a given program point. This refinement can facilitate other kinds of optimizations beside run-time check elimination. Moreover, in sharp contrast to traditional abstract interpretation systems, a direct implementation of the least flow graph generated by the constraint rules can be easily applied to analyze incrementally-defined programs.

Our results indicate that flow analysis offers a promising platform upon which to implement run-time check elimination. If the framework is equally adept in supporting other optimizations, we expect to use the ideas upon which it is based in an optimizing compiler for Scheme.

References

[1] Alexander Aiken and Wimmers, Edward and T.K Lakshman. Soft Typing with Conditional Types. In *21th ACM Symposium on Principles of Programming Languages*, pages 163–173, 1994.

[2] Henk Barendregt. *The Lambda Calculus.* North-Holland, Amsterdam, 1981.

[3] William Clinger and Jonathan Rees, editors. Revised^4 Report on the Algorithmic Language Scheme. *ACM Lisp Pointers*, 4(3), July 1991.

[4] Luis Damas and Robin Milner. Principle Type-schemes for Functional Programs. In 9th *ACM Symposium on Principles of Programming Languages*, pages 207–212, January 1982.

[5] Kent Dybvig. *The Scheme Programming Language.* Prentice-Hall, Inc., 1987.

[6] Marc Feeley, Marc Turcotte, and Guy Lapalme. Using MultiLisp for Solving Constraint Satistfaction Problems: An Application to Nucleic Acid 3D Structure Determination. *Lisp and Symbolic Computation*, 7(2/3):231–248, 1994.

[7] Leslie Greengard. *The Rapid Evaluation of Potential Fields in Particle Systems.* ACM Press, 1987.

[8] Christopher Haynes. Infer: A Statically-typed Dialect of Scheme. Preliminary Tutorial and Documentaion. Technical report, Indiana University, March 1993.

[9] Nevin Heintze. Set-Based Analysis of ML Programs. In *Proceedings of the ACM Symposium on Lisp and Functional Progmmming*, pages 306–317, 1994.

[10] Fritz Henglein. Global Tagging Optimization by Type Imerence. In *Proceedings of the ACM Symposium on Lisp and Functional Progmmming*, pages 205–215, 1992.

[11] R. Hindley. The Principal Type-Scheme of an Object in Combinatory Logic. *Transactions of the American Mathematical Society*, 146:29–60, December 1969.

[12] Suresh Jagannathan and Stephen Weeks. A Unified Treatment of Flow Analysis in Higher-Order Languages. In 22^{th} *ACM Symposium on Principles of Progmmming Languages*, pages 392–401, January 1995.

[13] Neil Jones and Stephen Muchnick. Flow Analysis and Optimization of Lisp-like Structures. In 6^{th} *ACM Symposium on Principles of Programming Languages*, pages 244–256, January 1979.

[14] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer und System Sciences*, 17:348–375, 1978.

[15] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

[16] John Mitchell. *Handbook of Theoretical Computer Science: Volume B*, chapter Type Systems for Programming Languages, pages 367–453. MIT Press, 1990.

[17] Jens Palsberg. Global Program Analysis in Constraint Form. In *Proceedings of the 1994 Colloquium on Trees in Algebra und Programming*, pages 276–290. Springer-Verlag, 1994. Appears as LNCS 787.

[18] Jens Palsberg and Patrick O'Keefe. A Type System Equivalent to Flow Analysis. In 22^{th} *ACM Symposium on Principles of Programming Languages*, pages 367–378, January 1995.

[19] Jens Palsberg and Michael Schwartzbach. Safety Analysis versus Type Inference. *Information und Computation, to appear.*

[20] Gordon D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[21] Olin Shivers. Data-flow Analysis and Type Recovery in Scheme. In *Topics in Advanced Language Implementation.* MIT Press, 1990.

[22] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages or Taming Lambda.* PhD thesis, School of Computer Science, Carnegie-Mellon University, 1991.

[23] Peter Steenkiste and John Hennessy. Tags and type checking in lisp. In *Proceedings of the Second Architectural Support for Progmmming Languages und Systems Symposium*, pages 50–59, 1987.

[24] Andrew Wright and Robert Cartwright. A Practical Soft Type System for Scheme. In *Proceedings of the ACM Symposium on Lisp und Functional Programming*, pages 250–262, 1994.

[25] Feng Zhao. An $O(N)$ Algorithm for Three-Dimensional N-Body Simulations. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1987.

Syntactic Type Polymorphism for Recursive Function Definitions

Jean-Pierre Talpin and Yan-Mei Tang
European Computer-Industry Research Centre and Ecole Des Mines de Paris

June 1993

Abstract

Higher-order programming languages, such as ML, permit a flexible programming style by using compile-time type inference together with the concept of type polymorphism, which allows to specify the types of generic functions. In ML, however, recursive factions must always be given a unique (monomorphic) type inside their defition. Giving polymorphic types to recursive functions is known as the problem of *polymorphic recurcion* which has been shown equivalent to the problem of semi-unification, known as undecidable. We show that the absence of a decidable specification to give polymorphic types for recursive definitions lies on the non-adequacy of representing type polymorphism by using type variables as primitive elements. We introduce the notion of *syntactic type polymorphism* to relate polymorphic types with syntactic information. We formulate a decidable calculus which gives polymorphic types to recursive factions in ML. We present an inference algorithm which we prove the termination and correctness.

1 Introduction

Higher-order programming languages, such as Standard ML [8], permit a flexible programming style by using the technique of compile-time type inference [7, 2] for type-checking programs. This technique is one of the most appreciated features in ML-like languages as it delegates to the compiler part of the task of the programmer to write extensive type declarations. This automation is not done by sacrificing the expressiveness of the type system.

Standard ML allows static typing of generic functions (e.g. functions which may uniformly operate on arguments of different types) by using the notion of *type polymorphism*. It has, however, a limitation. Recursive functions must always be given the same type inside their definition. Giving polymorphic types to recursive functions in ML is known as the difficult problem of *polymorphic recursion*.

Polymorphic recursion was first introduced in the Milner-Mycroft calculus [9]. Latter, deciding typing in this calculus was shown equivalent to the problem of semi-unification [5], which was then proved undecidable [6]. Meanwhile, decidable restriction of semi-unification were isolated as the ground for sound implementations of the Milner-Mycroft calculus [5].

The interest on the Mimer-Mycroft calculus has recently been renewed by the discovery of its application in the specification of program analysis techniques [15], such as effect systems [3, 12]. In this context, we are interested in introducing a decidable calculus for assigning polymorphic types to recursive functions which neither the syntax-directed processing of the Damas-Milner calculus nor the undecidable Milner-Mycroft calculus yet permit.

We show that the difficulty of giving a decidable account to generic recursive functions in the Damas-Milner calculus comes the fact that from using type variables as primitive elements is non-adequate for representing type polymorphism. Therefore, we introduce a notion of *syntactic type polymorphism* which relates polymorphic types with syntactic information. Using syntactic type polymorphism, we formulate a calculus which gives polymorphic types to recursive functions and present an inference algorithm which we prove the termination and correctness of.

2 Preliminary Examples

To address the problem *polymorphic recursion*, we consider some simple examples that establish the connection between mutually recursive function definitions and type polymorphism. Let us define f, g and h as follows.

$$f(x) = x : \alpha \to \alpha \qquad g(y) = f(3) : \alpha \to \mathsf{int} \qquad h(z) = j(\mathsf{true}) : \alpha \to \mathsf{bool}$$

Each of these definition is typable in the Damas-Milner calculus. If we want to give them a mutually recursive definition: "$\mathsf{rec}\ f(x) = x$ and $g(y) = f(3)$ and $h(z) = f(\mathsf{true})''$, we can not type them anymore. The erason lies on the fact that only one type of the function f must be given inside the recursive definition. This type cannot simultaneously match $\alpha' \to \mathsf{int}$ and $\alpha'' \to \mathsf{bool}$.

To overcome this problem, the principle of the Milner-Mycroft calculus is to assume a *type scheme* $\forall\alpha.\alpha \to \alpha$ of f, as a representation of all its types, and then prove that $\alpha \to \alpha$ is a valid type of f. The essence of this calculus is to solve a fixed-point equation between the assumption $\forall\alpha.\alpha \to \alpha$ and the result $\alpha \to \alpha$ of the proof. However, a straightforward iteration technique fails to terminate on other instances of the problem, such as: "$\mathsf{rec}\ f(g) = g(f)''$.

If we take $\forall\alpha_0.\alpha_0$ as an initial assumption for f and iterate, we first prove "$[f : \forall\alpha_0.\alpha_0] \vdash \mathsf{rec}\ f(g) = g(f) : (\alpha_1 \to \alpha_2) \to \alpha_2''$ where α_2 is given to $g(f)$ and $\alpha_1 \to \alpha_2$ to g and continue iteration, getting $\mathsf{rec}\ f(g) = g(f) : ((\alpha_3 \to \alpha_4) \to \alpha_4) \to \alpha_4$. We discover that each time we instantiate the type scheme of f in the body of its definition, we built a new function type with that instance as argument. Such a constraint does not have any finite representation. Therefore, we should say that f is untypable.

By showing that typing a program in the Milner-Mycroft calculus reduces to solving a problem in semi-unification, the technique employed in [5] allows discovering such a situation. It detects that there is a cyclic instantiation constraints between the type scheme of f and its instance $g(f)$ in the body of the function. This is done by performing an *extended occurrence-check* over a set of collected instantiation constraints. This technique allows to characterize a sound implementation of the Milner-Mycroft calculus.

Still, there are some other programs that exhibit cyclic instantiation constraints while being typable in the Milner-Mycroft calculus. For instance, the definition $\mathsf{rec}\ f(x) = (f(f))(x); x$.

It first applies f to f, then $f(f)$ to x and then returns x, can be given type $\forall\alpha.\alpha \to \alpha$ in the Milner-Mycroft calculus. But there is (following [5]) a cyclic instantiation constraint between $f : \alpha \to \alpha$ and $f(f) : (\alpha \to \alpha) \to (\alpha \to \alpha)$, represented by the semi-unification constraints $\alpha \to \alpha \le \alpha'$ and $\alpha \to \alpha \le \alpha' \to (\alpha \to \alpha'')$.

3 Approach

In a programming language, the specification of a static semantics aims at providing a tool to reason about properties of programs. By specification, we understand a set of axioms and rules which characterize a judgment of the form: "given the assumption a, the expression e has type t". Our contribution is to provide such a specification for assigning generic properties to recursive definitions.

We can easily observe that the representation of polymorphism using primitive or symbolic type variables a is not convenient for typing recursive functions. This formalism does not provide us with enough information about the program. Thus, the resulting specification does not provide enough discipline in the choice of types.

To allows us to precisely control the amount of type polymorphism introduced during the type-checking of a program, we choose to represent type variables in strong connection with the syntax of the program.The idea is to associate each type variables with the set of program labels at which this type variable is used.

Let us reuse the first example of the previous section, with a few program labels made explicit, to sketch how our technique works.

$$\mathsf{rec}\ f_a(x_b) = x \text{ and } g(y) = f_c(3) \text{ and } h(z) = f_d(\mathsf{true})$$

We want to give a generic type to the declared function f (in place a) and then give it exactly one (related) instance in the places it is called: $c : f(3)$ and $d : f(\mathsf{true})$. The idea is to compose the type of f with the label mentioned at the first place: $f : \forall\alpha_{a,b}.\alpha_{a,b} \to \alpha_{a,b}$ and then to aggregate this type with the labels found at the place it needs to be instantiated. It will have types $f_c : \alpha_{a,b,c} \to \alpha_{a,b,c}$ and $f_d : \alpha_{a,b,d} \to \alpha_{a,b,d}$.

We obtain exactly one instance of the type off at each place the type needs to be instantiated. Furthermore, each instance is related to the original (generic) type by a covariant extension of the set-membership relation on the structure of types (we assume that ground types are bigger than type variables).

$$\alpha_{a,b} \rightarrow \alpha_{a,b} \subseteq \alpha_{a,b,c} \rightarrow \alpha_{a,b,c} \subseteq \alpha_{a,b,c,e} \rightarrow \mathsf{int}$$
$$\alpha_{a,b} \rightarrow \alpha_{a,b} \subseteq \alpha_{a,b,d} \rightarrow \alpha_{a,b,d} \subseteq \alpha_{a,b,d,f} \rightarrow \mathsf{bool}$$

If now, we consider the second example, $\mathsf{rec}\ f_a(g_b) = g(f_c)_d$ we discover that the type of f has a component $\alpha_{a,b,c}$ which needs to match a bigger term $\alpha_{a,e,c} \rightarrow \alpha_{a,e,c,d}$.

A similar situation happens with the third example of the previous section. This shows that, by representing type variables in close connection with the syntax of the program, we can reduce the problem of determining the solution of a set of acyclic semi-unification constraints into a system of equations over sets of program labels.

4 Contribution

Using this notion of syntactic polymorphism, we formulate a calculus derived from that of [7, 2] which, in addition, allows giving polymorphic types to recursive functions, in the spirit of [9]. In our presentation, we systematize the use of inference rules, not only to specify the static semantics of our language, but also to define our unification and inference algorithms.

We obtain a calculus which give a decidable account to assigning polymorphic types for recursive functions. In the context of its applications to type-based program analysis or effect systems, it can serve as a basis to express generic properties for recursive definitions in a decidable and uniform framework. This approach offers significant advantages over previous investigations in this area [15]. It gives uniform results: it has a syntactically correct inference algorithm. It is simpler to reason with: it has a decidable logic.

Detailed proofs of the formal properties presented in this paper are available in [13] together with a proof that our calculus is consistent with respect to a dynamic semantics of the language. In particular, we introduce a proof technique derived from previous work in fixed-point theory [1] to establish the termination of our algorithm.

5 Syntax of the Language

We start by giving the syntax of our language. We consider a countable set of program labels p. Values v are either integers n, λ-abstractions, written $\mathsf{fn}(x_p) = e$, or recursive functions, written $\mathsf{rec}\ f_p(x_p) = e_p$. Identifiers i are either value identifiers x or identifiers of recursive function definitions f. Expressions are either values v, an identifiers i_p, applications $e(e)_p$ or a let. To each value and expression in a program corresponds a unique label p.

p	program labels
$v ::= n \mid \mathsf{fn}(x_p) = e \mid \mathsf{rec}\ f_p(x_p) = e_p$	values
$i ::= x \mid f$	identifiers
$e ::= v \mid i_p \mid e(e)_p \mid \mathsf{let\ val}\ x = e\ \mathsf{in}\ e$	expressions

Syntax

6 A Syntactic Type System

We define the quantities manipulated in our calculus by the following production rules. A *syntactic type variable* α is represented by a set of labels p assembled with the union operator "$\cup$". A *syntactic type* t is either a ground type int, a type variable α or a function type $t \to t$. We write $fv(t)$ the sequence α of type variables in t. A *type scheme* g, written $\forall\alpha.t$, is a type t quantified over the sequence of its free variables α. Type schemes are not equivalent under renaming of bound syntactic variables. We write $fv(\forall\alpha.t) = fv(t)\backslash\alpha$ and $bv(\forall\alpha.t) = \alpha$. An *assumption* is a sequence $a[i : g]$ of identifiers i and type schemes g, $a(i)$ is type scheme associated to i in a.

$$\begin{array}{ll} \alpha ::= \{p\} \mid \alpha \cup \alpha & \text{type variables} \\ t ::= \mathsf{int} \mid \alpha \mid t \to t & \text{types} \\ g ::= \forall \alpha . t & \text{generic types} \\ a ::= [] \mid a[i : g] & \text{assumptions} \end{array}$$

A syntactic type system

Types obey a covariant partial order relation $t \subseteq t'$. It denotes that the type t' can be used in strictly more places than t, either because t' contains either more program labels than t (for instance $t = \{p\}$ and $t' = \{p, p'\}$ or because it is a simpler types (for instance $t' = \{p, p'\} \to \{p, p'\}$ or $t' = \mathsf{int}$).

$$\alpha \subseteq \mathsf{int} \quad \alpha \subseteq \alpha \cup \alpha' \quad \frac{\alpha \subseteq t_1 \quad \alpha \subseteq t_2}{\alpha \subseteq t_1 \to t_2} \quad \frac{t_1 \subseteq t_1' \quad t_2 \subseteq t_2'}{t_1 \to t_2 \subseteq t_1' \to t_2'}$$

Specialization relation $t \subseteq t'$

A *substitution* is either the identity [] or, written $[t/\alpha]$ that replaces all free occurrences of the type variable α by a term t. A substitution is a strictly $\subseteq$-extensive morphism (e.g. either $\alpha = t$ or $\alpha \subset \alpha'$ for all $\alpha' \in fv(t)$). We write $s \circ s'$ for the composition of s and s' and s_a the restriction of a substitution s to the free variables of a. We identify $s(a)$ and $s_a(a)$.

7 Static Semantics

We use structured operational semantics to give an inductive definition of our static semantics as the judgment $a \vdash e \ : \ t$, which reads: "given the assumptions a, the expression e has type t". The structure of this specification is similar to previous polymorphic calculi [7, 2, 9] except that type variables contain program labels.

$$a \vdash n \;:\; \mathsf{int} \qquad \frac{t \preceq_p a(i)}{a \vdash i_p \;:\; t} \qquad \frac{a \vdash e \;:\; t \quad a[x \;:\; gen(a,t)] \vdash e' \;:\; t'}{a \vdash \mathsf{let\ val}\ x = e\ \mathsf{in}\ e' \;:\; t'}$$

$$\frac{a[x : t] \vdash e \;:\; t' \quad p \subseteq t}{a \vdash \mathsf{fn}(x_p) = e \;:\; t \to t'} \qquad \frac{a \vdash e \;:\; t' \to a \vdash e' \;:\; t' \quad p \subseteq t}{a \vdash e(e')_p \;:\; t'}$$

$$\frac{a[f \;:\; gen(a,t)] \vdash \mathsf{fn}(x_{p'}) = e \;:\; t \quad p \subseteq t \quad t' \preceq_{p''} gen(a,t)}{a \vdash \mathsf{rec}\ f_p(x_{p'}) = e_{p''} \;:\; t'}$$

Static semantics: $a \vdash e \;:\; t$

An instance t' of a type scheme $\forall\alpha.t$ at a program point p, written $t' \preceq_p \forall\alpha.t$, satisfies $t' = t[t/\alpha]$ where $\alpha \subseteq t$ and $p \subseteq t$. This means that the term substituted for all bound type variables in $\forall\alpha.t$ must mention p.

We write $gen(a,t)$ the *closure* of the type t over its *quantifiable* variables α. We write $gen(a,t) = \forall\alpha.t$ where $\alpha = \{\alpha \in fv(t) \mid \forall\alpha' \in fv(a), \alpha' \not\subseteq \alpha\}$. This means that a type variable α is generalizable if neither α nor a smaller $\alpha' \subseteq \alpha$ occur free in a. This avoids capture of α via the $\subseteq$-extensive substitution of α' by α.

8 Formal Properties

In this section, we characterize the invariants and formal properties of our calculus. We define the formal criterion $wf(a)$ which specifies how assumptions a are constructed in the static semantics. We write $wf([])$ and write $wf(a[i : g])$ if and only if $wf(a)$ and for all $\alpha \in bv(g)$, for all $\alpha' \in fv(a[i : g]), \alpha' \not\subseteq \alpha$. The property $wf(a)$ is stable under substitution $wf(s(a))$ and generalization $wf(a[i : gen(a,t)])$. Given the above, we prove that our syntactic calculus is stable under substitution on free variables and is preserved under stronger assumptions. We write $g' \preceq g$ if and only if, for any t and p, $t \preceq_p g$ implies $t \preceq_p g'$.

Lemma 1. *If* $a[i : g] \vdash e \;:\; t$ *and* $g' \preceq g$ *then* $a[i : g'] \vdash e \;:\; t$

Lemma 2. *If $wf(a)$ and $a \vdash e : t$ then $s(a) \vdash e : s(t)$*

9 Unification Algorithm

Constraint resolution plays a central role in the process of computing the principal type of programs. Thus, we start by defining our unification algorithm before to present the inference algorithm. It is sketched by a set of inference rules which associate a type equation "$t = t'$" to a $\subseteq$-extensive and idempotent substitution s which satisfy it: "$s(t) = s(t')$". When a type equation cannot be solved, the unifier returns fail_s.

$$[\alpha\cup\alpha'/\alpha][\alpha\cup\alpha'/\alpha'] \text{ solves } \alpha = \alpha' \quad \mathsf{fail}_s \text{ solves } \mathsf{int} = t \to t' \quad \mathsf{fail}_s \text{ solves } t = \mathsf{fail}_t$$

$$\begin{array}{l} [\mathsf{int}/\alpha] \text{ solves } \alpha = \mathsf{int} \\ [\mathsf{int}/\alpha] \text{ solves } \mathsf{int} = \alpha \end{array} \qquad \frac{s \text{ solves } t_1 = t_2 \quad s' \text{ solves } s(t'_1) = s(t'_2)}{s' \circ s \text{ solves } t_1 \to t'_1 = t_2 \to t'_2}$$

$$\frac{\alpha \in fv(t \to t')}{\begin{array}{l}\mathsf{fail}_s \text{ solves } \alpha = t \to t' \\ \mathsf{fail}_s \text{ solves } t \to t' = \alpha\end{array}} \qquad \frac{\alpha \notin fv(t \to t') \quad s = [\alpha \cup \alpha'/\alpha' \in fv(t \to t')]}{\begin{array}{l}s \circ [t \to t'/\alpha] \text{ solves } \alpha = t \to t' \\ s \circ [t \to t'/\alpha] \text{ solves } t \to t' = \alpha\end{array}}$$

Specification of the Unification Algorithm: s solves $t = t$

The inference rules are recursively defined on the structure of the input pair (t, t'). Computations are required in the case of function types, where s replaces all $\alpha' \in fv(t \to t')$ with $\alpha \cup \alpha'$, in order for the substitution of α by $s(t \to t')$ to be $\subseteq$-extensive. The correctness of the unification algorithm is stated as follows.

Lemma 3 *If s solves $t = t'$ then $s(t) = s(t')$. If $s'(t) = s'(t')$ then s solves $t = t'$ and there exists s'' such that $s'(t, t') = s''(s(t, t))$*

10 Inference Algorithm

In this section, we give the specification of our type inference algorithm using a set of inference rules, written $a \triangleright e : t\ (s)$, which, given a set of assumptions a, describes how the principal type t of an expression e is computed.

Unlike the static semantics, the algorithm associates untypable expressions with $(\mathsf{fail}_t, \mathsf{fail}_s)$. This pair is the supremum of all other pairs (t, s). We identify fail_t with $t \rightarrow \mathsf{fail}_t$, $\mathsf{fail}_t \rightarrow t$, $s(\mathsf{fail}_t)$ and $\mathsf{fail}_s(t)$ for any t, $\mathsf{fail}_s \circ s$ with fail_s for any s. We add that $t \subseteq \mathsf{fail}_t$ for any t. We use the rule $a[i : \mathsf{fail}_t] \triangleright e : \mathsf{fail}_t\ (\mathsf{fail}_s)$ to propagate errors. Instantiation is written $inst_p(\forall \alpha.t) = t[\alpha \cup \{p\}/\alpha]_{\alpha \in \alpha}$.

To infer the type of an abstraction, we take the program point p, at which the identifier x is mentioned, as initial assumption for its type. Then, we infer the type of the body and apply on p the resulting substitution s, which sums up all the typing constraints that p must verify. For the application, we introduce a type term, taken as the type of its result, which contains the current program point p and satisfy the constraint "$t' \rightarrow p$ equals $s'(t)$".

To determine the type of a recursive function definition, we perform a fixed-point iteration, which is specified by the sequence of judgments ranging over j. Initially, we give the type $t_0 = p$ to the function identifier. At each iteration j, we apply the previous substitutions s'_j and s'_j on a_j and then generalize the type $s'_j(t_j)$ to g_j to infer the type t_{j+1} and a substitution s_{j+1}. Then, we constraint t_{j+1} to be equal to $s_{j+1}(s'_j(t_j))$, getting the substitution s'_{j+1}. The iteration terminates upon satisfaction of "$s'_{k+1}(t_{k+1}) = s'_k(t_k)''$.

$$\frac{a \triangleright e \ : \ t\ (s) \quad s(a)[x \ : \ \mathrm{gen}(s(a),t)] \triangleright e' \ : \ t'\ (s'))}{a \triangleright \ \mathsf{let\ val}\ x = e\ \mathsf{in}\ e' \ : \ t'(s' \circ s)} \qquad \frac{a[x:p] \triangleright e \ : \ t\ (s))}{a \triangleright \mathsf{fn}(x_p) = e \ : \ s(p) \to t\ (s)}$$

$$a \triangleright i_p \ : \ inst_p(a(i))[] \qquad \frac{a \triangleright e \ : \ t\ (s) \quad s(a) \triangleright e' \ : \ t'\ (s)}{a \triangleright e(e')_p \ : \ s''(p)\ (s'' \circ s' \circ s)} \qquad (s'' \text{ solves } s'(t) = t' \to p)$$

$$\frac{\begin{array}{c}\forall j > 0, a_{j-1}[f : g_{j-1}] \triangleright \mathsf{fn}(x_{p'}) = e \ : \ t_j\ (s_j) \\ s'_j \text{ solves } t_j = s_j(s'_{j-1}(t_{j-1})) \\ a_j = s'_j(s_j(a_{j-1})) \text{ and } g_j = gen(a_j, s'_j(t_j))\end{array}}{a \triangleright \mathsf{rec}\ f_p(x_{p'}) = e_{p''} \ : \ inst_{p''}(g_{k+1}(\circ_{j \le k+1}(s'_j \circ s_j))} \qquad \left(\begin{array}{l} s'_0 = [], t_0 = p, a_0 = a \\ g_0 = gen(a_0, t_0) \\ s'_k(t_k) = s'_{k+1}(t_{k+1}) \end{array}\right)$$

Inference Algorithm:: $a \triangleright e \ : \ t(s)$

The unification performed during each iteration corresponds exactly to the notion of widening in abstract interpretation [1]). Indeed, one can observe that the relation "$s_j(t_{j-1}) \subseteq t_j$" does not always hold (for instance, $s_1(t_0) \not\subseteq t_1$). Therefore, we must use the unification algorithm to make sure that $s'_j(s_j(t_{j-1})) \subseteq s'_j(t_j)$ holds. This allows us to characterize our fixed-point iteration technique as a widening operation (e.g. a $\subseteq$-extensive operation) and give a simple proof of termination of our inference algorithm, section 12.

11 An Example Revisited

To illustrate how the inference algorithm determines the type of an expression, we reconsider the example of $\mathsf{rec}\ f_1(g_2) = g_3(f_4)_{5\ 6}$ which was given in the introduction. We label it with from 1 to 6. The inference algorithm proceeds the program using the set $\{\alpha_s \mid s \subseteq \{1,2,3,4,5,6\}\}$ of syntactic type variables. The first iteration of the algorithm, traced below, determines the "shape" of the type of the abstraction "$\mathsf{fn}(g_2) = g_3(f_4)_5$" given the initial assumption "$a_1 = [f : \forall \alpha_1.\alpha_1]$".

$$\frac{a_1[g:\alpha_2] \triangleright g_3 : \alpha 2 \quad a_1[g:\alpha_2] \triangleright f_4 \ : \ \alpha_4}{a_1[g:\alpha_2] \triangleright g_3(f_4)_5 \ : \ \alpha_{25}\ (s_1)} \qquad \text{where } s_1 \text{ solves } \alpha_2 = \alpha_{14} \rightarrow \alpha_5$$

$$\frac{a_1[g:\alpha_2] \triangleright g_3(f_4)_5 \ : \ \alpha_{25}\ (s_1)}{a_1 \triangleright \mathsf{fn}(g_2 = g_3(f_4)_5 \ : \ (\alpha_{124} \rightarrow \alpha_{25}) \rightarrow \alpha_{25}\ (s_1)} \qquad \text{because } s_1(\alpha_2) = (\alpha_{124} \rightarrow \alpha_{25})$$

$$\frac{a_1 \triangleright \mathsf{fn}(g_2) = g_3(f_4)_5 \ : \ (\alpha_{124} \rightarrow \alpha_{25}) \rightarrow \alpha_{25}\ (s_1)}{[] \triangleright \mathsf{rec} f_1(g_2) = g_3(f_4)_5 \ : \ (\alpha_{124} \rightarrow \alpha_{125}) \rightarrow \alpha_{125}\ (s_1' \circ s_1)}$$

where s_1' solves $\alpha_1 = (\alpha_{124} \rightarrow \alpha_{25}) \rightarrow \alpha_{25}$. The type $s_1'(t_1)$ given to f after the first iteration satisfies all collected constraints. We start the second iteration with it, taking as assumption

$$a_2 = [f : \forall\alpha_{124}.\forall\alpha_{125}.(\alpha_{124} \rightarrow \alpha_{125}) \rightarrow \alpha_{125}]$$

The algorithm then terminates, issuing that the expression is untypable. The reason is that the types t_1 and t_2 resulting from the iterations 1 and 2 cannot be unified. More insight on the trace of the constraint resolution algorithm permit to understand why:

$$\frac{a_2 \triangleright \mathsf{fn}(g_2) = g_3(f_4)_5 \ : \ (((\alpha_{124} \rightarrow \alpha_{1245}) \rightarrow \alpha_{25}) \rightarrow \alpha_{25}\ (s_2)}{[] \triangleright \mathsf{rec}\ f_1(g_2) = g_3(f_4)_5 \ : \ \mathsf{fail}_f\ (\mathsf{fail}_s)}$$

With $(s_2(t_1), t_2)$ as input, the resolution algorithm discovers a fixed-point equation between the type variable α_{124} and the term $((\alpha_{124} \rightarrow \alpha_{1245}) \rightarrow \alpha_{1245}) \rightarrow \alpha_{125}$.

$$\mathsf{fail}_s \text{ solves } \alpha_{124} = ((\alpha_{124} \rightarrow \alpha_{1245}) \rightarrow \alpha_{1245}) \rightarrow \alpha_{125}$$

This equation does not, of course, have a finite representation in our type system. The detection of this equation is enabled by limiting the number of instances of the type variable α_{124}, at program point 4, during the second iteration.

12 Decidability

We establish that our inference algorithm terminates. An important issue in this proof is to show that the fixed-point iteration technique employed to infer the type of recursive definitions terminates given finite inputs (a, e). To this end, we formulate the appropriate setting of fixed-point theory [11, 1]. The first step is to equip us with an ad-hoc extension of the relation $\subseteq$ to relate the successive results (t, s) of the iteration employed to infer the type of recursive definitions.

Definition 4. For any substitution s, we write $s \subseteq s'$ if and only if $s(\alpha) \subseteq s'(\alpha)$ for any α. For any pair (t, s), we write $(t, s) \subseteq (t', s')$ if and only if $t \subseteq t'$ and $s \subseteq s'$.

We determine the iterator $F_{a,e}$ employed in the inference algorithm to compute the type of recursive definitions. It takes the pair (t_j, s''_j) as argument and returns the pair (t_{j+1}, s''_{j+1}). Using $F_{a,e}$, we define the chain $X_{a,e}$ which is computed during the iteration.

Definition 5. Given the expression $\mathsf{rec}\ f_p(x_{p'}) = e'$, written e, and well-formed assumptions a, we define $F_{a,e}(t, s) = (s''(t'), s'' \circ s' \circ s)$ where $s(a)[f : gen(s(a), t)] \triangleright \mathsf{fn}(x_{p'}) = e' \ : \ t' \ (s')$ and s'' solves $s'(t) = t'$. The chain $X_{a,e} = (X_{a,e})_{j \geq 0}$ is defined by $(X_{a,e})_0 = (p, [])$ and by $(X_{a,e})_{j+1} = F_{a,e}(X_{a,e})_j$ for any $j \geq 0$

We first make the observation that $F_{a,e}$ is $\subseteq$-extensive. By application of Tarski's fixed-point theorem [11], $F_{a,e}$ admits a least-fixed point $\mathit{lfp}(F_{a,e})$ and the chain $X_{a,e} = (X_{a,e})_{j \geq 0}$ is strictly increasing for $\subseteq$. Then, we show that the iteration employed in the inference algorithm to compute $X_{a,e}$ terminates.

Lemma 6. *For any (t, s), $(t, s) \subseteq F_{a,e}(t, s)$*

Lemma 7. *The chain $X_{a,e}$ constructed using $F_{a,e}$ is finite on any input (a, e)*

The proof of lemma 7, in [13], relates the cardinality of $X_{a,e}$ to the variables constructed using $\mathit{fv}(a)$, $\mathit{bv}(a)$ and $\mathit{fp}(e)$. A careful reading of the definition of the algorithm allows to actually determine the number of iteration needed to compute the type of a recursive function f as being the number of occurrences of f in its definition. The only place in the expression e where new program points can be introduced in the pair $X_{a,e}$ are exactly these of occurrences f_p of the function identifier f in the body e of the function. We generalize the previous result to all expressions e by stating the termination theorem.

Theorem 8. *The derivation $a \triangleright e \ : \ t \ (s)$ teminates on all input (a, e)*

13 Correctness

We prove the syntactic correctness of our algorithm with respect to the static semantics. We first show that the result of the algorithm has a proof in the static semantics. Then, we state that any proof of the static semantics is weaker than the result of the algorithm. In other words, the algorithm is complete and computes the principal type of expressions. In particular, the type computed for recursive definitions is the least fixed-point of the operator $F_{a,e}$, section 12.

Theorem 9. *If $wf(a), a \triangleright e \ : \ t \ (s)$ and $(t, s) \neq (\mathsf{fail}_t, \mathsf{fail}_s)$ then $s(a) \vdash e \ : \ t$. If $wf(a)$ and $s(a) \vdash e \ : \ t$ then $a \triangleright e \ : \ t' \ (s')$ and there exists s'' such that $(s(a), t) = s''(s'(a), t')$*

14 A Classification of Recursive Definitions

Examples of recursive functions which require type polymorphism can be classified in two categories. The first category, previously mentioned in [9] and [5] are mutually recursive definitions of the form $\mathsf{rec}\ f_1(x_1) = e_1 \ldots \mathsf{and}\ f_n(x_n) = e_n$ where the identifiers $f_{1,\ldots n}$ occur in the expressions $e_{1\ldots n}$ with different and non matching types. Such examples cannot be typed in the Damas-Milner calculus. As seen previously, they can be typed using the Milner-Mycroft calculus. They can, in our calculus, as long as exactly one type is required for typing each occurrence of $f_{1,\ldots n}$ in $e_{1,\ldots n}$.

In our calculus, the typable examples from this first category have in common that each instance t_i^j of the types t_i of the recursive function identifiers

f_i satisfies $t_i \subseteq t_i^j$. There is, however, another category of recursive functions which could benefit from a type system with polymorphic recursion. Examples in this category can be constructed by considering sophisticated user data-types, where one of the constructor takes a strictly bigger object than the one returned. An example is the following definition.

```
datatype a' Tree = Leaf | Node of 'a * (('a * 'a) Tree)

fun flatten t =
    case t of
         Leaf => []
         | Node (a, p)) => a ::  (map fst (flatten p))
                               @  (map snd (flatten p))
```

The constructor Node has type $\alpha \times ((\alpha \times \alpha)$ Tree $\rightarrow \alpha$ Tree. We define a recursive function to flatten the data-structure. This function collects from an α Tree the elements of type α at its leaves. It has type α Tree $\rightarrow \alpha$ list in the Milner-Mycroft calculus.

In our calculus (provided an extension of the static semantics to handle pattern matching), the recursive call needs type f : $(\alpha \times \alpha)$ Tree $\rightarrow (\alpha \times \alpha)$ list which cannot match that of flatten. The reason is that all program labels at which flatten is instantiated are accumulated into the type of a and p (of type $(\alpha \times \alpha)$ Tree). In the algorithm of [5], a type error is similarly reported, resulting of an unsolvable instantiation constraint.

15 Related Work

As suggested from the results of all the examples presented in this paper, our approach to type polymorphism for recursive functions provides very similar results to the approach presented in [5]. The approach of Henglein is, in essence, algorithmic: in [5], the author shows that typability in the Milner-Mycroft calculus is equivalent to solving a problem of semi-unification. He presents an algorithm which derives from a program a set of semi-unification constraints. This set of semi-unification constraints is complete in the sense that each Milner-Mycroft typable expression is a solution to this set of semi-unification constraints.

In spite of the fact that the problem of semi-unification has been shown undecidable [6], Henglein has shown that acyclic semi-unification constraints could be solved. This provided the basis for a syntactically sound implementation of the Mimer-Mycroft calculus. Having shown the equivalence between typability in the Mimer-Mycroft calculus and semi-unification allows one to consider a wide spectrum of techniques to solve ever more sophisticated semi-unification problems.

Some (unpublished) are reported in [16]. The first approach consist to iterate over a set of semi-unification constraints with a limited number of type variables (as in our calculus). This enforces termination but, of course, does not suffice to formally characterize the fixed point of solvable constraints (like we do in section 12). Another consist to test whether the system of instantiation inequations fall into a decidable class of semi-unification problems (or make sense as a matching statement) and to solve it if this is the case. Reportedly, this approach allows typing some programs in the second category of the previous section while we cannot.

Our approach differs in its principles to previous and present work on polymorphic recursion. Instead of giving a better algorithm to solve more typing problems in the Milner-Mycroft calculus, we propose a calculus which gives a limited but decidable account on assigning generic properties to recursive definitions. Our prime interest for such a development is to provide a basis for defining program analysis techniques based on principles of typing: effect systems. In this context, our approach offers signmcant advantages over previous investigations in this area [15].

The analysis presented in [15] uses a variant of the Milner-Mycroft calculus to express generic data-flow properties in recursive definitions. The inference algorithm uses the semi-unification algorithm of [5]. Since this algorithm fails on cyclic instantiation constraints, it must sometimes give a conservative (monomorphic) account to the properties of certain recursive functions. This has two disadvantages.

Practically speaking, cyclic instantiation constraints occur in function definitions which employ sophisticated recursion mechanisms. Giving a monomorphic approximation of their properties seriously impacts the quality of the analysis and of dependent program optimizations. Theoretically speaking, it requires two logics to reason with. One with polymorphic properties for recursive definitions (with respect to which the inference algorithm is sound) and one with monomorphic properties for recursive definitions (with respect to which the inference algorithm is complete).

On the opposite, our approach is more uniform in its results: an inference algorithm of data-usage properties based on the principle of syntactic polymorphism never fails for well-typed expressions. It is simpler to reason with: it has one logic and a syntactically correct inference algorithm.

16 Formal Relation to Other Calculi

In this section, we establish a formal comparison of our calculus with respect to the Damas-Milner and Milner-Mycroft calculus. To this end, we give its definition and then define the criteria which helps relating it to ours. First of all, we must consider an hypothesis which is always used in the type systems based on the Damas-Milner calculus. It is that type-schemes are equivalent under renaming of bound type variables. Instead of this hypothesis, we make the equivalent assumption that all type variables bound in assumptions Γ are distinct. This is written $Wf_{\mathsf{DM}}(\Gamma)$.

The property *wf* must be preserved under generalization and instantiation. We write σ a type scheme $\forall\beta.\tau$ and $\tau \preceq_{\mathsf{DM}} \Gamma(i)$ any τ such that $\Gamma(i) = \forall\beta.\tau'$, $\tau = \tau'[\tau/\beta]$ where $fv(\tau) \cap bv(\Gamma) = \emptyset$. We write $\overline{\Gamma}(\tau) = \forall\beta.\tau[\beta/\beta']$ where $\beta' fv(\Gamma) fv(\tau)$ and $\beta \cap (fv(\Gamma) \cup bv(\Gamma)) = \emptyset$. We relate typing judgments using a one to one map m from symbolic type variables β and syntactic type variables α. We write $m(\beta)$ the α associated to β by m. We show that any well-formed Damas-Milner proof has a proof in our calculus via a map m. Our calculus is thus complete with respect to the Damas-Milner calculus.

Proposition 10. *If* $Wf_{\mathsf{DM}}(\Gamma)$ *and* $\Gamma \vdash_{\mathsf{DM}} e : \tau$ *then there exists* m *such that* $m \models (\Gamma, \tau) : (a, t)$, $wf(a)$ *and* $a \vdash e : t$

The converse statement is false. This can be shown by considering the example: "$\mathsf{rec}\ f_1(x_2) = (f_3(7); f_4(\mathsf{true}); x)$". It is not typable using the Damas-Milner calculus and has type $\alpha_{12} \rightarrow \alpha_{12}$ in our calculus. But we can similarly show that any proof in our calculus can be related to a one in the Milner-Mycroft calculus.

Proposition 11. *If* $wf(a)$ *and* $a \vdash e \ : \ t$ *then there exists* $Wf_{\mathsf{DM}}(\Gamma)$ *and* $m \models (\Gamma, \tau) : (a, t)$ *such that* $\Gamma \models_{\mathsf{MM}} e \ : \ \tau$

Again, the converse statement is false. Our calculus is more restrictive that the Milner-Mycroft calculus in the generalization and the instantiation of bound recursive function types. For instance, the program "$\mathsf{rec}\ f_1(x_2) = (f_3(f_4)f_5)(x)_6; x$" has type $\forall\alpha.\alpha \to \alpha$ in the Milner-Mycroft calculus, but cannot be assigned a type in our calculus.

17 Conclusion

We have presented a decidable calculus for giving polymorphic types to recursive functions in ML. We have proved its consistency with respect to the dynamic semantics of ML and introduced correct inference and constraint resolution algorithms to implement it. We have conducted a systematic formalization of the calculus and of its implementation using structured operational semantics. We have introduced a proof technique adapted from previous results of fixed-point theory to prove the termination of our algorithm.

As mentioned in the introduction, we are primarily interested in providing a calculus which can serve as a basis for defining program analysis[1]. To this respect, it has been demonstrated [15] that expressing properties of recursive functions truly requires polymorphism. In this paper, we have presented a decidable account and a syntactically correct inference technique for describing generic properties of recursive definitions. In a forthcoming paper [14], we show that this framework can successfdy be applied to analysing alias and lifetime of structured data, obtaining an effect system which favorably competes over all previous type-based analysis techniques.

[1]More than proposing an extension to the type system of Standard ML with polymorphic recursion. Such an extension seems to be of limited interest compared to other approaches, such as explicit type polymorphism, which may well address this very issue in a simpler way

18 Implementation Issues

Before concluding, we would like to give some hints on some important issue in efficiently implementing our calculus. The first is to choose an appropriate representation for syntactic type variables and the second to delimit the scope within which they must be used. A first observation is that, in a modular program, this scope is that of expressions in the module language (structure and signature objects in Standard ML).

At this level, variables can be *exported* (by a process similar to *signature matching* in Standard ML), by guarantying that a renaming exists between an inferred type (in a structure) and a declared or exported type (in the signature). Type information can be reported to the user by using the same scheme. Conversely, values which we know the signature can be *imported* within the analyser by using a reverse process. To represent syntactic type variables, we use a hash-table to relate symbolic data with syntactic data. This guarantees unique allocation of every syntactic type variables.

19 Acknowledgments

This work was partially funded by the Basic Research Action No. 8130 LOMAPS. The authors thank the program committee for the numerous and valuable comments they provided [16].

References

[1] Cousot, P. Semantic foundation of program analysis. In (Muchnick and Jones, editors) *Program flow analysis, theory and application*, chapter 10, pages 303–342. Prentice-Hall Inc., 1981.

[2] Damas, L., and Milner, R. Principal type schemes for functional programs. In *Proceedings of the 1982's ACM Symposium on Principles of Programming Languages*, January 1982.

[3] Gifford, D. K., Jouvelot, P., Lucassen, J. M., and Sheldon, M. A. FX-87 Reference Manual. *MIT/LCS/TR-407*, MIT Laboratory for Computer Science, September 1987.

[4] Harper, R. A simpl*Technical Report 93-169*, School of Computer Science, Carnegie Mellon University. June 1993.

[5] Henglein, F. Type inference with polymorphic recursion. In *ACM Transaction on Programming Languages and Systems*, vol. 15(2), pages 253–289. ACM Press, April 1993.

[6] Kfoury, A.J., Tiuryn, J., and Urzyczyn, P. Type recursion in the presence of polymorphic recursion. In *ACM Transaction on Programming Languages and Systems*, vol. 15(2), pages 290–311. ACM Press, April 1993.

[7] Milner, R. A Theory for type polymorphism in programming. In *Journal of Computer and Systems Sciences*, Vol. 17, pages 348–375. 1978.

[8] Milner, R., Tofte, M., Harper, R. The definition of Standard ML. *The MIT Press*, 1989.

[9] Mycroft, A. Polymorphic type schemes and recursive detitions. In *Proceedings of the 6th. International Conference on Programming.* Lectures Notes in Computer Science No. 167. Springer Verlag, 1984.

[10] Robinson, J. A. A machine oriented logic based on the resolution principle. In *Journal of the ACM*, Vol. 12(1), pages 23–41. ACM Press, 1965.

[11] Tarsky, A. A lattice theoretical moint theorem and its apphcations. In *Pacific Journal of Mathematics*, volume 5, pages 285–309. 1955.

[12] Talpin, J.-P., and Jouvelot, P. The type and effect discipline. In *Information and Computation*, Vol. 110. Academic Press, 1994.

[13] TaIpin, J.-P., and Tang, Y.-M. Syntactic Type Polymorphism for Recursive Function Defitions. *Technical Report ECRC-94-29.* European Computer-Industry Research Centre, July 1994. Available from `http://www.ecrc.de/ ~jp/home.html`.

[14] Talpin, J.-P., and Tang, Y.-M. Principles of syntactic subtyping and iterative inference techniques for determining the Iife-time of data-regions in functional programs. *Technical Report ECRC-95-16.* European Computer-Industry Research Centre, April 1995. Available soon from `http://www.ecrc.de/ ~jp/home.html.`

[15] Tofte, M., and Talpin, J.-P. Implementation of the typed lambda-calculus using a stack of regions. In *Proceedings of the 1994's ACM Symposium on Principles of Programming Languages*, pages 188–201. ACM Press.

[16] Communications of the TPA'95 workshop program committee.