Fast Binding-Time Analysis for Multi-Level Specialization

Robert Glück[1] and Jesper Jørgensen[2]

[1] University of Copenhagen, Dept. of Computer Science, Universitetsparken 1
DK-2100 Copenhagen, Denmark, e-mail: glueck@diku.dk

[2] Katholieke Universiteit Leuven, Dept. of Computer Science, Celestijnenlaan 200a
B-3001 Heverlee, Belgium, e-mail: jesper@cs.kuleuven.ac.be

Abstract. Program specialization can divide a computation into several computation stages. We present the key ingredient of our approach to multi-level specialization: an accurate and fast *multi-level binding-time analysis*. Three efficient program analyses for higher-order, functional languages are presented which are based on *constraint systems* and run almost-linear in the size of the analyzed programs. The three constraint normalizations have been proven correct (soundness, completeness, termination, existence of best solution). The analyses have all been implemented for a substantial, higher-order subset of Scheme. Experiments with widely-available example programs confirm the excellent run-time behavior of the normalization algorithms.

Keywords: program transformation, partial evaluation, program analysis, generating extensions, functional languages.

1 Introduction

The division of programs into *two stages* has been studied intensively in partial evaluation and mixed computation to separate those program expressions that can be safely preevaluated at specialization time from those that cannot. The main problem with the binding-time analysis of standard *partial evaluation*, *e.g.* as presented in [13], is the need to specify the availability of data in terms of 'early' (*static*) and 'late' (*dynamic*). This two-point domain does not allow to specify multi-level transition points (*e.g.* "dynamic until stage n"). This has limited the operation of partial evaluators to a conservative two-level approximation. Our goal is more general: *multi-level specialization*.

This paper presents the key ingredient of our approach to multi-level specialization: an accurate and fast *multi-level binding-time analysis*. We introduce a general binding-time domain that expresses different 'shades' of static input. This means that a given program can be optimized with respect to some inputs at an earlier stage, and others at later stages. This modification requires several non-obvious extensions of standard partial evaluation techniques, such as *multi-level generating extensions* [8], a generalization of Ershov's (two-level) generating extension [6]. The main payoff of this novel approach becomes apparent in multiple self-application: experimental results show an impressive reduction of generation time and code size compared to previous attempts of multiple self-application.

The analyses we present are constraint-based. *Constraint-based program analysis* has only been developed in this decade, among others, out of the desire to speed-up program analysis for partial evaluation. The main motivation is efficiency: type-inference based monovariant binding-time analysis can be done by constraint normalization in almost-linear time [11] (in contrast to abstract interpretation based methods). Traditionally, binding-time analysis has been formulated in the framework of *abstract interpretation* (*e.g.* [5, 3]) and *type-theoretic* settings (*e.g.* [15, 11, 10]).

Our multi-level binding-time analysis has the same accuracy as and is slightly faster than the two-level analysis in Similix [4], a state-of-the-art partial evaluator, which is notable because we did not optimize our implementation for speed. The results are also significant because they clearly demonstrate that multi-level specialization scales up to advanced languages without performance penalties.

The analyses presented in this paper are not particularly biased towards partial evaluation. A distinction between binding-times (*e.g.* compile-time vs. run-time), flow of function values, and partial type inference are, among others, important for an efficient implementation and transformation of functional languages. Our work builds on previous work on constraint-based analysis [4] and multi-level specialization [8].

This paper. After reviewing the concept of multi-level generating extensions (Sect. 2) we present the typing-rules for multi-level programs of a higher-order, functional language (Sect. 3), and we introduce three efficient program analyses based on *constraint systems* that run almost-linear in the size of the analysed programs (Sect. 4). The three analyses together give an algorithmic solution for the multi-level binding-time analysis specified by the typing rules. Experiments with widely-available example programs confirm the excellent run-time behavior of the normalization algorithms (Sect. 5).

2 Generating Extensions

We summarize the concept of multi-level generating extensions. The notation is adapted from [13]: for any program text, `p`, written in language L we let $[\![\texttt{p}]\!]_{\text{L}}\,\texttt{in}$ denote the application of the L-program `p` to its input `in`.

Ershov's Generating Extensions. A program generator `cogen`, which we call *compiler generator* for historical reasons, is a program that takes a program `p` and its binding-time classification (bt-classification) as input and generates a program generator `p-gen`, called *generating extension* [6], as output. The task of `p-gen` is to generate a residual program `p-res`, given static data $\texttt{in}_0$ for `p`'s first input. We call `p-gen` a *two-level* generating extension of `p` because it realizes a two-staged computation of `p`. A generating extension `p-gen` runs potentially much faster than a program specializer because it is a specialized program generator.

$$\begin{aligned}\texttt{p-gen} &= [\![\texttt{cogen}]\!]_{\text{L}}\ \texttt{p 'SD'}\\ \texttt{p-res} &= [\![\texttt{p-gen}]\!]_{\text{L}}\ \texttt{in}_0\\ \texttt{out} &= [\![\texttt{p-res}]\!]_{\text{L}}\ \texttt{in}_1\end{aligned}$$

Multi-Level Generating Extensions. Program specialization can do more than stage a computation into two phases. Suppose `p` is a source program with n inputs. Assume the input is supplied in the order $\texttt{in}_0 \ldots \texttt{in}_{n-1}$. Given the first input $\texttt{in}_0$ a multi-level generating extension produces a new specialized multi-level generating extension $\texttt{p-mgen}_0$ and so on, until the final value `out` is produced from the last input $\texttt{in}_{n-1}$. Multi-level specialization using multi-level generating extensions is described by

$$\begin{aligned}
\texttt{p-mgen}_0 &= [\![\texttt{mcogen}]\!]_{\mathrm{L}}\ \texttt{p}\ \text{`}0 \ldots n{-}1\text{'} \\
\texttt{p-mgen}_1 &= [\![\texttt{p-mgen}_0]\!]_{\mathrm{L}}\ \texttt{in}_0 \\
&\vdots \\
\texttt{p-mgen}_{n-2} &= [\![\texttt{p-mgen}_{n-3}]\!]_{\mathrm{L}}\ \texttt{in}_{n-3} \\
\texttt{p-res}'_{n-1} &= [\![\texttt{p-mgen}_{n-2}]\!]_{\mathrm{L}}\ \texttt{in}_{n-2} \\
\texttt{out} &= [\![\texttt{p-res}'_{n-1}]\!]_{\mathrm{L}}\ \texttt{in}_{n-1}
\end{aligned}$$

Our approach to multi-level specialization is *purely off-line*. A program generator `mcogen`, which we call a multi-level compiler generator, or short multi-level generator, is a program that takes a program `p` and a bt-classification $t_0 \ldots t_{n-1}$ of `p`'s input parameters and generates a *multi-level generating-extension* [8] $\texttt{p-mgen}_0$. The order in which input is supplied is specified by the bt-classification. The smaller the bt-value t_i, the earlier the input becomes available.

It is easy to see that a standard (two-level) generating extension is a special case of a multi-level generating extension: it returns only an 'ordinary' program and never a generating extension. Programs $\texttt{p-mgen}_{n-2}$ and `p-gen` are examples of two-level generating extensions.

3 Multi-Level Binding-Time Analysis

We are going to develop a *multi-level binding-time analysis* (MBTA) for the multi-level generator `mcogen` in the remainder of this paper. The task of a MBTA is briefly stated: given a source program p, the binding-time values (bt-values) of its input parameters together with a maximal bt-value, find a consistent multi-level annotation of p which is, in some sense, the 'best' in some sense. We give typing rules that define well-annotated multi-level programs and specify the analysis.

Multi-Level Language. The multi-level language is an annotated, higher-order subset of Scheme [12] where every construct has a bt-value $t \geq 0$ as additional argument. In addition the multi-level language has a lift operator $\underline{\texttt{lift}}^s_t$ (Fig. 1). The underlining of an operator, *e.g.* $\underline{\texttt{if}}_t$, together with the bt-value t attached to it, is its *annotation*. Consistency and minimality of annotations will be defined below. Function calls are always unfolded so no annotation is used.

Not all multi-level programs have a consistent bt-annotation. We give typing rules that formalize the intuition that early values may not depend on late values. They define *well-annotated* multi-level programs. Before we give the set of rules, we explain bt-values and bt-types. For every source program p the bt-values t_i of its input parameters are given, as well as a maximal bt-value **max**.

Definition 1 (binding-time value). A *binding-time value* (bt-value) is a natural number $t \in \{0, 1, \ldots, \mathbf{max}\}$ where **max** is the *maximal bt-value* for the given problem.

A *binding-time type* τ contains information about the type of a value, as well as the bt-value of the type. The bt-value of an expression e in a multi-level program is equal to the bt-value $\|\tau\|$ of its bt-type τ. In case an expression is well-typed (*wrt* a monomorphic type system with recursive types and one common base type), the type component of its bt-type τ is the same as the standard type.

Definition 2 (binding-time type). A type τ is a (well-formed) binding-time type *wrt* **max**, if $\vdash \tau{:}t$ is derivable from the rules below. If $\vdash \tau{:}t$ then the type τ *represents* a bt-value t, and we define a mapping $\|\cdot\|$ from types to bt-values: $\|\tau\| = t$ **iff** $\vdash \tau{:}t$.

$$\{Base\}\ \ \Delta \vdash B^t{:}t \qquad \{Fct\}\ \ \frac{\Delta \vdash \tau_i{:}s_i \quad \Delta \vdash \tau{:}s \quad s_i \geq t \quad s \geq t}{\Delta \vdash \tau_1 \ldots \tau_n \rightarrow^t \tau{:}t}$$

$$\{Btv\}\ \ \frac{\alpha{:}t \text{ in } \Delta}{\Delta \vdash \alpha{:}t} \qquad \{Rec\}\ \ \frac{\Delta \oplus \{\alpha{:}t\} \vdash \tau{:}t}{\Delta \vdash \mu\alpha.\tau{:}t}$$

Base bt-types, shown in Rule $\{Base\}$, are denoted by B^t where t is the bt-value. We do not distinguish between different base types, *e.g.* integer, boolean, *etc.*, since we are only interested in the distinction between base values and functions. Rule $\{Fct\}$ for function types requires that the bt-values of the argument types $\tau_1 \ldots \tau_n$ and the result type τ are *not* smaller than the bt-value t of the function itself because *before* the function is applied neither the arguments are available to the function's body nor can the result be computed. Rule $\{Btv\}$ ensures that the bt-value t assigned to a type variable α^t is never greater than **max**. Rule $\{Rec\}$ for recursive types $\mu\alpha.\tau$ states that τ has the same bt-value t as the recursive type $\mu\alpha.\tau$ under the assumption that the type variable α has the bt-value t. The notation $\Delta \oplus \{\alpha : t\}$ denotes that the bt-environment Δ is extended with $\{\alpha : t\}$ while any other assignment $\alpha : t'$ is removed from Δ. This is in accordance with the equality $\mu\alpha.\tau = \tau[\mu\alpha.\tau/\alpha]$ which holds for recursive types.

An *equivalence relation* on bt-types allows us to type *all* expressions in our source language even though the language is dynamically typed. In particular, we can type expressions where the standard types cannot be unified because of potential type errors (function values used as base values, base values used as function values). We defer such errors to the latest possible binding time.

Definition 3 (equivalence of bt-types). Let **max** be a maximal bt-value and let U be the following equation:

$$B^{\mathbf{max}} \ldots B^{\mathbf{max}} \rightarrow^{\mathbf{max}} B^{\mathbf{max}} \;=\; B^{\mathbf{max}}$$

Given two bt-types τ and τ' well-formed *wrt* **max**, we say that τ and τ' are equivalent, denoted by $\vdash \tau \doteq \tau$', if $\tau = \tau'$ is derivable from equation U, the equivalence of recursive bt-types (based on types having the same regular type, and the usual relation for equality (symmetry, reflexivity, transitivity, and compatibility with arbitrary contexts).

Typing Rules. The typing rules for well-annotated multi-level programs are defined in Fig. 1. Most of the typing rules are generalizations of the corresponding rules used for two-level programs in partial evaluation, *e.g.* [13]. For instance, rule $\{If\}$ for `if`-expressions annotates the construct `if` with the bt-value t of the test-expression e_1 (the `if`-expression is reducible when the result of the test-expression becomes known at time t). The rule also requires that the test expression has a first-order type.

Rule $\{Lift\}$ shows the multi-level operator $\underline{\texttt{lift}}_t^s$: the value of its argument e has bt-value t, but its results is not available until $t+s$ $(s>0, t\geq 0)$. The bt-value of an expression $(\underline{\texttt{lift}}_t^s\ e)$ is the sum of the bt-values s and t. In other words, the multi-level lift operator delays a value to a later binding time. As is customary in partial evaluation, the rule allows lifting of first-order values only.

Rule $\{Op\}$ requires that all higher-order arguments of primitive operators have bt-value **max**. This is a necessary and safe approximation since we assume nothing about the type of a primitive operator (the same restriction is used in the binding-time analysis of Similix [3]).

$$\{Con\}\ \Gamma \vdash c{:}B^0 \qquad \{Var\}\ \frac{x{:}\tau \text{ in } \Gamma}{\Gamma \vdash\ x{:}\tau}$$

$$\{If\}\ \frac{\Gamma \vdash e_1{:}B^t \quad \Gamma \vdash e_2{:}\tau \quad \Gamma \vdash e_3{:}\tau \quad \|\tau\| \geq t}{\Gamma \vdash (\underline{\texttt{if}}_t\ e_1\ e_2\ e_3){:}\tau} \qquad \{Call\}\ \frac{\Gamma \vdash e_i{:}\tau_i \quad f{:}\tau_1 ... \tau_k \rightarrow \tau \text{ in } \Gamma}{\Gamma \vdash (f\ e_1 ... e_k){:}\tau}$$

$$\{Let\}\ \frac{\Gamma \vdash e{:}\tau \quad \Gamma\{x{:}\tau\} \vdash e'{:}\tau' \quad \|\tau'\| \geq \|\tau\|}{\Gamma \vdash (\underline{\texttt{let}}_{\|\tau\|}\ ((x\ e))\ e'){:}\tau'} \qquad \{Op\}\ \frac{\Gamma \vdash e_i{:}B^t}{\Gamma \vdash (\underline{op}_t\ e_1 ... e_k){:}B^t}$$

$$\{Abs\}\ \frac{\Gamma\{x_i{:}\tau_i\} \vdash e{:}\tau'}{\Gamma \vdash (\underline{\lambda}_t\ x_1 ... x_n.e){:}\tau_1 ... \tau_n \rightarrow^t \tau'} \qquad \{App\}\ \frac{\Gamma \vdash e_0{:}\tau_1 ... \tau_n \rightarrow^t \tau' \quad \Gamma \vdash e_i{:}\tau_i}{\Gamma \vdash (e_0\ \underline{@}_t\ e_1 ... e_n){:}\tau'}$$

$$\{Lift\}\ \frac{\Gamma \vdash e{:}B^t \quad s{>}0}{\Gamma \vdash (\underline{\texttt{lift}}_t^s\ e){:}B^{t+s}} \qquad \{Equ\}\ \frac{\Gamma \vdash e{:}\tau \quad \vdash \tau \doteq \tau'}{\Gamma \vdash e{:}\tau'}$$

Fig. 1. Typing rules for well-annotated multi-level programs.

Definition 4 (well-annotated completion, minimal completion). Given a program p, a maximal bt-value **max**, and a bt-pattern $t_1 ... t_k$ of p's goal function f_0, a *well-annotated completion* of p is a multi-level program p' with $|p'| = p$ if the following judgment can be derived:

$$\vdash p'{:}\{f_0 : B^{t_1} ... B^{t_k} \rightarrow B^{\mathbf{max}}, f_1 : \tau_{11} ... \tau_{1n_1} \rightarrow^{t_1} \tau_1, ..., f_n : \tau_{n1} ... \tau_{nn_n} \rightarrow^{t_n} \tau_n\}$$

A well-annotated completion is *minimal* if the bt-value of every subexpression e in p is less than or equal to the bt-value of e in any other well-annotated completion of p.

Every program p has at least one well-annotated completion p' since the operations of a program can always be annotated with **max**, which corresponds to all subexpressions in the completion having the bt-type $B^{\mathbf{max}}$. A program p can have have more than one well-annotated completion. The goal of the MBTA is to determine a well-annotated completion p' which is, preferably, 'minimal', *i.e.* all operations in a program shall be performed as early as possible.

4 Analysis by Solving Constraints

Constraint-based program analysis is a method for efficiently solving certain classes of type inference problems. A *constraint system* C is a set of constraints $cs(t_1, ..., t_n)$ on terms $t_1, ..., t_n$ that may contain constraint variables. A constraint systems is solved by a series of *rewritings* into a *normalized system* that can be solved immediately (easier than the initial constraint system). The solution is obtained by composing the value substitution that solves the normalized constraint system with (the composition of) the elementary substitutions generated (and applied) during rewriting.

Notation. Every program point will be associated with a number of unique *constraint variables*. Constraint variables may range over different kinds of information, for instance bt-values. To denote that a constraint variable is associated with expression e, we index the variable: v_e. Similarly, a constraint variable associated with a program variable x is denoted by v_x. Furthermore, given a function definition `(define` $(f\ x_1 ... x_n)$ e`)`, we refer with v_{f_i} and v_f to the constraint variables v_{x_i} and v_e respectively.

Constraints and Domains. Three types of constraint variables occur in our constraint system.

Flow variables ϕ range over finite tuples $\pi\ \beta\ \langle \phi_1 ... \phi_n \rangle$ of constraint variables $(0 \leq n)$. We are dealing with a dynamically typed, higher-order language, so we need to be able to describe potential flow of functions into the *same* program point. Flow constraints capture the flow of values in a given program.

Projection-error variables (pj-variables) π range over pj-values $p \in \{B, Cl_1, ..., Cl_k, U\}$, a partially ordered set where $\sqsubseteq$ is the partial order induced by $B \sqsubseteq U$, $Cl_i \sqsubseteq U$. The top value is U ('untyped'). The pj-value B describes values of base type. A family of pj-values Cl describes function types of arity $0 \leq i \leq k$ where the maximum arity k is limited by the abstractions/applications occurring in a given program. User-defined n-ary constructors can be handled in essentially the same way as functions (omitted due to lack of space).

Binding-time variables (bt-variables) β range over bt-values b (integers) where $\leq$ is the usual (total) order on integers.

Capturing Binding-Time Rules by Constraints. The key observation is that the requirement that a multi-level program is well-annotated can be expressed as a set of constraints on the types of the individual subexpressions. These constraints can be efficiently reduced to a normal form from which the minimal completion is easily determined. This has been described in [11] where more details are given.

Let us give an intuitive idea of how constraints capture the bt-type system. Every derivation in the bt-type system corresponds uniquely to a labeling of the syntax tree with bt-types. The general principle is that first for each type rule one introduces variables for all types and adds side conditions to compensate for the loss of type conditions.

If there are several type rules that apply to the same syntactic form, one combines these rules into a single rule adding all their side conditions. This ensures that there is always exactly one rule that applies to any given syntactic form and the problem of writing down a type derivation becomes syntax directed. The side conditions then become the constraints of a constraint system.

There are two rules (Fig. 1) that cause overlap between rules: rule $\{Lift\}$ and rule $\{Equ\}$. Since repeated application of either one of these rule can always be replaced by a single application of the rule, we assume that in a derivation the application of a "proper" rule is followed by at most one application of the $\{Lift\}$ rule and at most one application of the $\{Equ\}$ rule. Since the $\{Lift\}$ rule changes the bt-value of an expression we introduce a new variable $\overline{\beta}_e$ for each expression e that represents the bt-value of the expression after lifting and add a *lift*-constraint $\beta_e \leq \overline{\beta}_e$ to the set of generated constraints for all such variables.

Constraint Generation. Figure 2 lists for each syntactic form of the source language the set of constraints that are generated. The division of the constraints into three groups will be explained below (Sect. 4.1). We implicitly assume "for all indices" when writing constraints where some parts have index subscripts; index ranges are implicitly defined by the context. For example, constraint $v_{e_i} = v_e$ actually stands for a series of constraints parameterized over e_i.

In our case the user can specify that a primitive operator *op* is not going to be performed before a certain binding time (specified by b_{op}). Usually all primitives of Scheme will have $b_{op} = 0$, which means that they can be performed as soon as all arguments are available, and only operators which may perform side-effects have $b_{op} = \mathbf{max}$.

Expression (e)	Flow constraints	Pj-constraints	Bt-constraints
c	$\pi_e \beta_e \langle\rangle \prec_B \phi_e$	$B \sqsubseteq \pi_e$	–
x	$\phi_e = \phi_x$	$\pi_e = \pi_x$	$\beta_e = \beta_x$
$(\underline{\texttt{if}}_{\beta_{e_1}}\ e_1\ e_2\ e_3)$	$\phi_e = \phi_{e_2}, \phi_e = \phi_{e_3}$	$\pi_e = \overline{\pi}_{e_2}, \pi_e = \overline{\pi}_{e_3}$ $B \sqsubseteq \overline{\pi}_{e_1}$	$\beta_e = \overline{\beta}_{e_2}, \beta_e = \overline{\beta}_{e_3}$ $\overline{\beta}_{e_1} \leq \beta_e$
$(\underline{\texttt{let}}_{\beta_{e_1}}((x\ e_1))\ e_2)$	$\phi_e = \phi_{e_2}, \phi_x = \phi_{e_1}$	$\pi_e = \overline{\pi}_{e_2}, \pi_x = \overline{\pi}_{e_1}$	$\beta_e = \overline{\beta}_{e_2}, \beta_x = \overline{\beta}_{e_1}$ $\overline{\beta}_{e_1} \leq \beta_e$
$(\underline{op}_{\beta_e}\ e_1 ... e_k)$	$\phi_e = \phi_{e_i}$	$\pi_e = \overline{\pi}_{e_i}, B \sqsubseteq \overline{\pi}_e$	$\beta_e = \overline{\beta}_{e_i}, b_{op} \leq \beta_e$
$(f\ e_1 ... e_k)$	$\phi_e = \phi_f, \phi_{e_i} = \phi_{f_i}$	$\pi_e = \overline{\pi}_f, \overline{\pi}_{e_i} = \pi_{f_i}$	$\beta_e = \beta_f, \overline{\beta}_{e_i} = \beta_{f_i}$
$(\underline{\lambda}_{\beta_e} x_1 ... x_n . e')$	$\pi_e \beta_e \langle \phi_{x_1} ... \phi_{x_n}, \phi_{e'} \rangle \prec_{Cl_n} \phi_e$	$Cl_n \sqsubseteq \pi_e$	$\beta_e \leq \beta_{x_i}, \beta_e \leq \overline{\beta}_{e'}$
$(e_0\ \underline{@}_{\overline{\beta}_{e_0}} e_1 ... e_n)$	$\overline{\pi}_{e_0} \overline{\beta}_e \langle \phi_{e_1} ... \phi_{e_n}, \phi_e \rangle \prec_{Cl_n} \phi_e$	$Cl_n \sqsubseteq \overline{\pi}_{e_0}$	$\overline{\beta}_{e_0} \leq \overline{\beta}_{e_i}, \overline{\beta}_{e_0} \leq \beta_e$

Fig. 2. Constraint generation.

Initial Constraints. In addition to the constraints generated from expressions in the program, the following constraints are part of the initial constraint system C_{init}. We assume that a program always inputs and outputs values of base type (as in the Similix partial evaluator).

1. For each input variable x_i of the goal procedure, a constraint $B \sqsubseteq \pi_{x_i}$ to ensure that the input is of base type and a constraint $b_i \leq \beta_{x_i}$ where b_i are the binding times for the input supplied by the user.
2. For the body e of the goal procedure, a constraint $B \sqsubseteq \pi_e$ and $\mathbf{max} \leq \beta_e$, since specialization never returns a value before the latest binding-time **max**.

3. For each expression e, a "lift" constraint $\pi_e \sqsubseteq \overline{\pi}_e$ and $\beta_e \leq \overline{\beta}_e$.

The "lift" constraints $\pi_e \sqsubseteq \overline{\pi}_e$ acts as barrier between expressions. A technique that can be used in general to 'weaken' equality-based constraint systems and to make them less conservative.

Constraint Normalization. Any solution to the initial constraint set C_{init} corresponds to a well-annotated completion and a typing derivation. The completion is easily obtained through the connection between subexpressions and binding-time variable expressed in Figure 2 and by inserting lift expression for expressions e where $\beta_e < \overline{\beta}_e$. The connection to the derivation is not really relevant for the finding of the completion, but the type of each subexpression e of the program can be found by applying the following translation to ϕ_e:

$$\begin{array}{ll} [\![B\ \beta\ \langle\rangle]\!] & = B^{\beta} \\ [\![U\ \beta\ \langle\phi_1, ..., \phi_n\rangle]\!] & = B^{\mathbf{max}} \\ [\![Cl_n\ \beta\ \langle\phi_1, ..., \phi_n, \phi\rangle]\!] & = [\![\phi_1, ..., \phi_n]\!] \rightarrow^{\beta} [\![\phi]\!] \end{array}$$

The derivation can then easily be reconstructed by insertion of $\{Equ\}$ rules to make the types fit.

Figure 3 shows the normalisation rules for the constraint system. We use $c@cs\ldots$ when we need to refer to constraint $cs\ldots$ as a whole and we can then simply refer to it using c. To find a solution to the initial constraint set C_{init} this is rewritten until no rule applies. The obtained *normal form constraint set* will then be in a form where a solution can be found directly.

Equality constraints of the form $v = rhs$ are always solved by substituting rhs for constraint variable v everywhere in the constraint system; that is, the *elementary substitution* $\sigma = [v \mapsto rhs]$ is applied to the constraint system. The solution ρ is obtained by composing the value substitution that solves the normalized constraint system with (the composition of) the elementary substitutions σ_i generated (and applied) during rewriting.

Minimal Solution. The normalized constraint set contains constraints that are trivially satisfied and for each expression e at most one constraint of the form $b \leq \beta_e$ and at most one constraint of the form $b' \leq \overline{\beta}_e$. This means that to find the minimal solution we can just interpret these constraints as equality constraints and thereby read of the solution for the binding-time analysis. The well-annotated completion is then easily constructed form this.

4.1 Ordered Constraint Normalization

The constraints of C_{init} can be solved in any order. But notice, for example, that flow constraints $\prec_{cl_n}$ are never generated by the application of the pj- and mbt-rules. Thus, all flow constraints are solved after exhaustively applying the flow rules (F_1, F_2) and the flow rules never needs to be considered again. Similarly, pj-constraints $\sqsubseteq$ are never generated by the application of the mbt-rules. Thus, all pj-constraints are solved after exhaustively applying the flow- and pj-rules (F_1, F_2, $P_1 ... P_4$).

The constraint normalization rules (Fig 3) can therefore be decomposed into three analysis where each exploits the results of the previous one. They are performed sequentially in the following order:

Flow constraint normalization:

$$F_1 : C \cup \{\phi_e = \phi_{e'}\} \qquad \Rightarrow [\phi_e \mapsto \phi_{e'}]C$$

$$F_2 : C \cup \left\{ \begin{array}{r} c@\pi_e\beta_e\langle\phi_{e_1}...\phi_{e_n}\rangle \prec_\chi \phi \\ \pi_{e'}\beta_{e'}\langle\phi_{e'_1}...\phi_{e'_n}\rangle \prec_\chi \phi \end{array} \right\} \Rightarrow C \cup \{c, \pi_e = \pi_{e'}, \beta_e = \beta_{e'}, \phi_{e_i} = \phi_{e'_i}\}$$

Pj-constraint normalization:

$$P_1 : C \cup \{\pi_e = \pi_{e'}\} \qquad \Rightarrow [\pi_e \mapsto \pi_{e'}]C$$

$$P_2 : C \cup \{c@p \sqsubseteq \pi_e, c'@\pi_e \sqsubseteq \pi_{e'}\} \Rightarrow C \cup \{c, c', p \sqsubseteq \pi_{e'}\}, \text{if } \forall p' \sqsubseteq \pi_{e'} \in C : p' \sqsubset p$$

$$P_3 : C \cup \{p \sqsubseteq \pi_e, p' \sqsubseteq \pi_e\} \qquad \Rightarrow C \cup \{p \sqcup p' \sqsubseteq \pi_e\}$$

$$P_4 : C \cup \{c@U \sqsubseteq \pi_e\} \qquad \Rightarrow C \cup \{c, \mathbf{max} \leq \beta_e\}$$

Bt-constraint normalization:

$$M_1 : C \cup \{\beta_e = \beta_{e'}\} \qquad \Rightarrow [\beta_e \mapsto \beta_{e'}]C$$

$$M_2 : C \cup \{c@b \leq \beta_e, b'@\beta_e \leq \beta_{e'}\} \Rightarrow C \cup \{b, b', p \leq \beta_{e'}\}, \text{if } \forall b' \sqsubseteq \beta_{e'} \in C : b' \sqsubset b$$

$$M_3 : C \cup \{b \leq \beta_e, b' \leq \beta_e\} \qquad \Rightarrow C \cup \{b \sqcup b' \leq \beta_e\}$$

Fig. 3. Constraint normalization.

1. **Flow analysis:** traces possible flow between abstractions and function applications.
2. **Projection error analysis**: determines possible type constructor conflicts.
3. **Binding-time analysis:** finds the (minimal) bt-value for each expression.

Moreover, a separate pj-analysis is necessary in order to reduced the mbt-analysis to a graph reachability problem that can be solved in linear time.

Correctness Proofs. For each analysis, we have proved that constraint normalization is sound and complete, that it terminates, and we have constructively proved existence of a minimal solution (substitution) to the initial constraint system. This paper does not prove that the initial constraint sets are correctly specified. Such proofs exist for specializers for the pure lambda-calculus (*e.g.* [21]). Although the language treated in this paper is much more complex, we found it easier to verify the analysis due to the separation into three orthogonal constraint systems.

5 Fast Normalization Algorithms

We give a short description of the implementation of the analyses. We use a union/find-based algorithm that operates on a *term graph representation with equivalence classes*. The implementation is based on the algorithm described in [4]; see also [11]. Actually, the algorithms are a simpler than those in [4] due to a decomposition of the analysis into three component analysis. The input to

each analysis is a constraint set generated by a one-pass compositional run over the program being analyzed.

The three analysis together give a solution for the MBTA specified by the typing rules in Fig. 1. The three component analyses have been implemented in Scheme and run slightly faster than the corresponding two-level analysis in the distribution version of Similix [4]. The overall analysis is monovariant since all calls to the same function have the same bt-classification.

(i) *Union/find based algorithm.* Constraint variable are represented by variable nodes in the term graph. Each constraint variable points to another variable node which is the *equivalence class representative* (*ecr*). The *ecr* represents equivalence classes of variable nodes generated by substitution rules. There are two nodes operations available on *ecr*'s: $\mathit{find}(n)$ for node n returns the *ecr* of the equivalence class to which n belongs; $\mathit{union}(n, n')$ which can only be applied to two distinct *ecr*'s n, n' merges the equivalence classes of both *ecr*'s and returns an *ecr* of the merged class. We shall not go into details with efficient union/find data structures, but refer to the literature; *e.g.* [19].

(ii) *Ordered constraint normalization.* Constraints are solved in a particular order (Sect. 4). For example, notice that equality constraints are never generated during constraint normalization of the pj- and mbt-analysis; thus they can be solved during the creation of the term graph.

Complexity. The normalization algorithms of the three analyses are based on the algorithms described in [4], thus their time complexities can be deduced by the same argument. The time complexities of the three analyses are linearly bounded by the size N of a program, modulo a small factor $\alpha(N, N)$[3].

Performance. Our experiments confirm that the analyses have an almost-linear behavior. It is notable that their total run-time is slightly faster than the corresponding two-level analysis in the distribution version of Similix. We conclude: multi-level analysis of programs has *no* penalty whatsoever (Figure 4 and 5).

Figure 4 gives the time and space consumption of the three analyses performed on selected programs. All programs are realistic examples: Bawl0 and Bawl1 are interpreters for a substantial lazy functional language [14], Spec and Cogen are the specializer and the automatically generated compiler generator of Similix, and Int is the two-level interpreter of [7]. Cogen was the largest program with 15814 cons cells.

All programs, except Int, were analysed with two levels (0,1) in order to compare our performance figures with those of the corresponding analyses in the Similix preprocessor [4]. The program Int, written in a way that its three inputs are properly separated *wrt* to binding-times, was analysed with three bt-levels (0,1,2).

Figure 5 gives the time and space consumption of the mbt-analysis performed on Int with different numbers of bt-level. One level: all inputs have mbt-value 0; two levels: all input have mbt-value 0 except one which has 1; *etc.* Note that

[3] $\alpha(N, N)$ is an inverse of the Ackermann's function that for all practical purposes (N's) is less than 4.

the performance of the analysis is not particularly influenced by more levels; no influence on storage consumption.[4]

Program	Size	Flow	PJ	MBT	Total	Similix
	cells	s/Kb	s/Kb	s/Kb	s/Kb	s/Kb
Bawl0	1420	.23/84	.26/91	.32/74	.81/249	.86/169
Spec	2089	.38/180	.40/153	.49/135	1.2/468	1.6/317
Bawl1	4602	.82/295	.93/304	1.1/261	2.9/860	3.0/579
Cogen	15814	2.7/1200	2.8/1038	3.4/893	8.8/3131	10/2148
Int	1195	.31/95	.86/175	.42/143	1.6/413	–

Fig. 4. Performance: component analyses

Levels	1	2	3	4
	s/Kb	s/Kb	s/Kb	s/Kb
MBT	.41/143	.42/143	.43/143	.46/143

Fig. 5. Performance: binding-time levels

6 Related Work

In partial evaluation the first constraint-based analysis for a higher-order language appears to be the one for the lambda-calculus [11]. An 'all-in-one' approach has been used for constraint-based analysis, *e.g.* [11, 13, 1, 2]; the *only* exception being the two-phase analysis [4]. For instance, the partial evaluator for the lambda-calculus [13] uses a single constraint-based analysis that performs the three tasks in a single run. A monomorphic type system with explicit binding-time annotations for the simply typed lambda-calculus was given in [15].

Partial type inference for the lambda-calculus was suggested in [9] and the corresponding algorithm for the inference of partial types was derived from the well-known algorithm W (that can be implemented in time $O(N^3)$; see [9] for more details). We use an almost-linear constraint-based algorithm for partial type inference in the pj-analysis that achieves essentially the same effect. Recently a polymorphic type system for dynamic typing of Scheme was devised [18] that is closely related to the pj-analysis. Data-flow analysis has be shown equivalent to type inference [17].

Closely related work has been done recently: our approach has been extended to continuation-based partial evaluation [20] and an algebraic description of multi-level lambda-calculi has been given in [16].

[4] All tests were run on a Sun 4/75 (SparcStation 2)/Sun OS 4.1.3 using Chez Scheme Version 3.2. Run times do not include garbage collection; storage allocation is measured separately. Size is measured as the number of "cons" cells needed to represent the program as an S-expression).

Acknowledgments. The second author was partly by the European HCM Network "Logic Program Synthesis and Transformation" and the Belgian GOA "Non-Standard Applications of Abstract Interpretation". The work was also supported by the project "Design, Analysis and Reasoning about Tools" funded by the Danish Natural Sciences Research Council.

References

1. L.O. Andersen. Binding-time analysis and the taming of C pointers. In *Partial Evaluation and Semantics-Based Program Manipulation*, ACM Press, 1993.
2. L. Birkedal, M. Welinder. Binding-time analysis for Standard ML. *Lisp and Symbolic Computation*, 8(3): 191–208, 1995.
3. A. Bondorf. Automatic autoprojection of higher order recursive equations. *Science of Computer Programming*, 17(1-3):3–34, 1991.
4. A. Bondorf, J. Jørgensen. Efficient analyses for realistic off-line partial evaluation. *J of Functional Programming*, 3(3):315–346, 1993.
5. C. Consel. Binding time analysis for higher order untyped functional languages. In *Conference on Lisp and Functional Languages.* 264–272, ACM Press, 1990.
6. A. P. Ershov. On the essence of compilation. In E. J. Neuhold (ed.), *Formal Description of Programming Concepts*, 391–420. North-Holland, 1978.
7. R. Glück, J. Jørgensen. Generating optimizing specializers. In *IEEE International Conference on Computer Languages*, 183–194. 1994.
8. R. Glück, J. Jørgensen. Efficient multi-level generating extensions for program specialization. In M. Hermenegildo, S. D. Swierstra (eds.) *Programming Languages, Implementations, Logics and Programs*, LNCS 982, 259–278, Springer-Verlag, 1995.
9. C. K. Gomard. Partial type inference for untyped functional programs. In *ACM Conference on Lisp and Functional Programming*, 282–287, ACM Press, 1990.
10. C. K. Gomard, N. D. Jones. A partial evaluator for the untyped lambda calculus. *J of Functional Programming*, 1(1):21–69, 1991.
11. F. Henglein. Efficient type inference for higher-order binding-time analysis. In J. Hughes (ed.), *Functional Programming and Computer Architecture*, LNCS 523, 448–472, Springer-Verlag, 1991.
12. IEEE. Standard for the Scheme programming language, IEEE Std 1178-1990.
13. N. D. Jones, C. K. Gomard, P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, 1993.
14. J. Jørgensen. Generating a compiler for a lazy language by partial evaluation. In *Symposium on Principles of Programming Languages.* 258–268, 1992.
15. F. Nielson, H. R. Nielson. *Two-Level Functional Languages*, volume 34 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 1992.
16. F. Nielson, H. R. Nielson. Multi-level lambda-calculus: an algebraic description. In O. Danvy, R. Glück, P. Thiemann (eds.), *Partial Evaluation.* LNCS 1110, Springer-Verlag, 1996.
17. J. Palsberg, P. O'Keefe. A type system equivalent to flow analysis. *TOPLAS*, 17(4):576–599, 1995.
18. J. Rehof. Polymorphic dynamic typing. Master's thesis, Univ. of Copenhagen, Denmark, 1995.
19. R. Tarjan. *Data Structures and Network Flow Algorithms*, vol. CMBS 44 of *Regional Conference Series in Applied Mathematics.* SIAM, 1983.
20. P. Thiemann. Cogen in six lines. In *International Conference on Functional Programming.* 180–189, ACM Press, 1996.
21. M. Wand. Specifying the correctness of binding-time analysis. *J of Functional Programming*, 3(3):365–387, 1993.

This article was processed using the LaTeX macro package with LLNCS style