Efficient computation of strictness types[*]

Fritz Henglein
DIKU, University of Copenhagen
Universitetsparken 1
DK-2100 Copenhagen East, Denmark
Email: henglein@diku.dk

March 4, 1994

Abstract

Amtoft has formulated an "on-line" constraint normalization method for solving a strictness inference problem inspired by Wright. From the syntactic form of the normalized constraints he establishes that every program expression has a unique, most precise ("minimal") strictness judgement, given fixed values for the strictness annotation in negative position.

We show that his on-line normalization is not required for proving his main syntactic result. Instead of normalizing the constraints during a bottom-up pass through the program we simply collect them first and solve them by standard iterative fixed point methods in a second phase. The main result follows from the fact that observable negative strictness variables only occur on the right-hand sides of the constraints. Furthermore, a standard iterative fixed point algorithm solves the constraints in linear time in the number of strictness variables in the constraint system whereas Amtoft's method requires exponential time.

Our presentation is somewhat different than Amtoft's. We use a linear-logic inspired presentation of the programming language and the strictness inference system for it. This results in smaller constraint systems generated. A tight strictness inference normalization result shows that many inequational constraints can be replaced by equational ones. Thus the constraint system can be simplified and significantly reduced in size by efficient variable unification, which can be performed on-line during its construction. Finally, we give an asymptotic worst-case analysis of the size of constraint systems relative to a program with or without explicitly typed variable declarations.

Generally, our method provides insight into the strengths of noncompositional program analysis methods; *i.e.*, methods where the solution for a program is not computed as a function of the corresponding solutions of its immediate components, such as in Algorithm W or in frontier-based abstract interpretation.[1] More specifically, our method demonstrates the effective use of efficient iterative fixed point computation known from classical data flow analysis in type-based program analysis.

1 Introduction

Amtoft has formulated an inference system for proving strictness properties and used it to translate call-by-name evaluation of a simple functional language to call-by-value evaluation [Amt93]. The inference system is a restriction of Wright's strictness formalization [Wri91]. It

[*]DART memo.
[1]Note that only the algorithmic method is noncompositional, not the program analysis itself!

can be seen to be essentially equivalent in expressive power to Baker-Finch's strictness analysis [BF92] when restricted to the language of the simply typed λ-calculus.

Amtoft's inference system is a subtying discipline with no minimal typing property. He shows, though, that given strictness annotations (*strict* or *nonstrict*) for all the *negative* positions in a *strictness judgement* there is a unique most precise assignment of strictness properties to all the *positive* positions in his system. He shows how his analysis can be reduced to solving inequalities between Boolean *strictness variables* and monotone Boolean expressions over these variables. A value of 0 for a strictness variable indicates "definitely strict" and 1 is "possibly nonstrict". He keeps the constraints in a certain syntactic form by *normalizing* them on-line whilst adding new constraints during a bottom-up pass through the subject program expression. The fact that the strictness variables in positive position have a unique minimal solution as a function of the variables in negative position follows from the particular form of the normalized constraints.

The constraints arising directly from the program constructs are of the form $b \geq s$ where s is a monotone Boolean expression over $\wedge, \vee$. The normalization steps preserve all the solutions for the variables, with one notable exception: The Park Induction principle

$$d \geq f(d) \Rightarrow d \geq \mu f$$

states that any postfixed point d of continuous (and thus monotone) function f is greater than the least fixed point μf of f. Amfoft uses this principle to rewrite recursive constraints of the form $\vec{b} \geq g(\vec{b})$ to $\vec{b} \geq \mu g$ where $\vec{b}$ ranges over $\mathcal{B}^k$, k-tuples of Boolean values, and g is a pointwise monotone function expression on such tuples. Since $\mathcal{B}^k$ has strictly ascending chains of length at most k we have $\mu g = \bigvee_{i=0}^{k} g^i(\vec{0})$ for any function g on Boolean k-tuples. Instead of using an explicit fixed point operator μ Amtoft uses the *expression* $\bigvee_{i=0}^{k} g^i(\vec{0})$ instead of μg.

There are two problems with this transformation:

1. The converse of the Park Induction principle does not hold in general. Consider, for example, $\mathit{swap}(b_1, b_2) = (b_1, b_2)$. Note that $(0,1) \geq \mu(\mathit{swap}) = (0,0)$, but we do not have $(0,1) \geq \mathit{swap}(0,1) = (1,0)$. Thus the transformation does not preserve the *set* of *all* solutions. Indeed it *adds* new solutions, making the transformation — at least in principle — unsound. Consequently Amtoft adds a new symbol $\gg$ and rewrites $\vec{b} \geq g(\vec{b})$ to $\vec{b} \gg \mu g$. The symbol has no fixed interpretation but is interpreted as either $=$ or $\geq$. A somewhat complicated relation between the solutions of the transformed constraints — under either interpretation — to the solutions of the original constraints is then used to argue the ultimate correctness of the normalization process.

2. The expression $\bigvee_{i=0}^{k} g^i(\vec{0})$ is vastly bigger than μg. In order to avoid superexponential blow-up the expressions are kept in conjunctive normal form during normalization. Nonetheless, in the worst case such an expression is of size exponential in the number m of strictness variables. As a consequence the inference algorithm executes in time $\Omega(2^m)$.

In this paper we show that there is no need to normalize constraints on the fly. Indeed this complicates the matter. By simply collecting all the constraints we obtain a constraint system of size *linear* in the number of strictness variables, with constraints of form $b \geq s$ where b is a strictness variable and s is a monotone Boolean expression. The result that positively occurring strictness variables have a least solution given values for the negatively occurring strictness variables is an immediate consequence of the following observation: negative strictness variables occur only on the right-hand side of constraints, positive strictness variables only on the left-hand side. Strictness variables not occurring at all in the final strictness judgement may occur on either or both sides of constraints.

$$\frac{\Gamma, x:\tau, y:\tau', \Delta \vdash e:\tau''}{\Gamma, y:\tau', x:\tau, \Delta \vdash e:\tau''} \quad \text{(EXCH)} \qquad\qquad \frac{\Gamma, x:\tau, x:\tau, \Delta \vdash e:\tau''}{\Gamma, x:\tau, \Delta \vdash e:\tau''} \quad \text{(CONTR)}$$

$$\frac{\Gamma, \Delta \vdash e:\tau'}{\Gamma, x:\tau, \Delta \vdash e:\tau'} \quad \text{(WEAK)} \qquad\qquad x:\tau \vdash x:\tau \qquad \text{(TRIV)}$$

$$\frac{\Gamma_x, x:\tau \vdash e:\tau'}{\Gamma_x \vdash \lambda x:\tau.e:\tau \to \tau'} \quad \text{(ABSTR)} \qquad\qquad \frac{\Gamma \vdash e:\tau \to \tau' \qquad \Delta \vdash e':\tau}{\Gamma, \Delta \vdash ee':\tau'} \quad \text{(APPL)}$$

$$\frac{\Gamma_x, x:\tau \vdash e:\tau}{\Gamma_x \vdash \mathbf{fix}\, x:\tau.e:\tau} \quad \text{(FIX)}$$

$$\vdash \mathit{true}:\mathsf{Bool} \qquad (\text{CONST}_{\mathit{true}}) \qquad\qquad \vdash \mathit{false}:\mathsf{Bool} \qquad (\text{CONST}_{\mathit{false}})$$

$$\frac{\Gamma \vdash e:\mathsf{Bool} \qquad \Delta \vdash e':\tau \qquad \Delta \vdash e'':\tau}{\Gamma, \Delta \vdash \mathbf{if}\, e\, e'\, e'':\tau} \quad \text{(COND)}$$

Figure 1: Static typing rules for functional language

Given values for the negatively occurring strictness variables the least solution for the positively occurring strictness variables is the restriction of the least solution of the whole constraint system to the positively occurring strictness variables. The constraints can be rewritten such that every variable occurs at most once on the left of a constraint. The least solution of C is the least fixed point of C understood as a system of equations (instead of inequations). Furthermore the least solution can be computed in time *linear* in the number of strictness variables by straightforward iterative fixed point computation.

2 A simply typed functional language

The object language is a simply typed higher-order functional language with Bool and possibly other primitive atomic types, but no structured types. For the sake of simplicity we restrict ourselves to only Booleans. The *types* are formal expressions produced by

$$\tau ::= \mathsf{Bool} \mid \tau \to \tau.$$

The language is given by the type inference system in Figure 1. We use the following conventions: τ ranges over types, x, y over variables, e over program expressions, and Γ, Δ over sequences of assumptions of the form $x : \tau$. This extends to sub- or superscripted metavariables. We write Γ_x for assumptions that do *not* contain an assumption for variable x.

This particular presentation explicitly contains the "structural" (in the logical meaning of the word) rules of exchanging assumptions (EXCH), contracting equal assumptions (CONTR), and

$$\Gamma\{x:\tau\} \vdash x:\tau \qquad \text{(VAR)}$$

$$\frac{\Gamma\{x:\tau\} \vdash e:\tau'}{\Gamma \vdash \lambda x:\tau.e:\tau \to \tau'} \ \text{(ABS)} \qquad \frac{\Gamma \vdash e:\tau \to \tau' \qquad \Gamma \vdash e':\tau}{\Gamma \vdash ee':\tau'} \ \text{(APP)}$$

$$\frac{\Gamma\{x:\tau\} \vdash e:\tau}{\Gamma \vdash \mathbf{fix}\, x:\tau.e:\tau} \ \text{(FIX)}$$

$$\Gamma \vdash \mathit{true}:\mathsf{Bool} \qquad (\text{CON}_{\mathit{true}}) \qquad\qquad \Gamma \vdash \mathit{false}:\mathsf{Bool} \qquad (\text{CON}_{\mathit{false}})$$

$$\frac{\Gamma \vdash e:\mathsf{Bool} \qquad \Gamma \vdash e':\tau \qquad \Gamma \vdash e'':\tau}{\Gamma \vdash \mathbf{if}\, e\ e'\ e'':\tau} \ \text{(IF)}$$

Figure 2: Alternative static typing rules for functional language

weakening by adding an additional assumption (WEAK). Strictness properties are intimately related to the use of (WEAK), as noted by Baker-Finch, since weakening introduces an *irrelevant* hypothesis; *i.e.*, a hypothesis for a variable x, which is not used in the deduction of the type of an expression and thus cannot have any influence on the evaluation of the expression under call-by-name.

A more conventional presentation of the language is given in Figure 2, where environments are finite *maps*, and $\Gamma\{x:\tau\}$ is defined by

$$\Gamma\{x:\tau\}(y) = \begin{cases} \Gamma(y), & \text{if } x \neq y \\ \tau, & \text{if } x = y \end{cases}$$

It can be shown *for closed expressions* e that $\vdash e:\tau$ by Figure 1 if and only if $\emptyset \vdash e:\tau$ by Figure 2. This does not hold for open expressions, though. This is due to the fact that we might have two or more *different* type assumptions for the same (free) variable x in Figure 1 whereas there is always at most one assumption for any variable in Figure 2. Multiple different assumptions can never be contracted to a single assumption and thus we cannot close an expression that requires multiple different assumptions for a variable by λ- or **fix** -abstraction. Put another way, if $\Gamma \vdash e':\tau'$ occurs in a derivation of $\vdash e:\tau$ then Γ is *consistent*: if $x:\tau', x:\tau'' \in \Gamma$ then $\tau = \tau'$.

As we shall see, strictness analysis — as indeed many other type-based program analyses — is defined by induction on the standard type derivation for an expression. We choose Figure 1 as our point of departure instead of Figure 2 since:

- the rules for program constructs in Figure 1 express naturally strictness properties directly and the structural rules can be annotated to show how strictness properties are combined;

- the constraints for variable occurrences we shall construct in Section 5 are of size proportional to the judgement $x:\tau \vdash x:\tau$. In this case Amtoft generates constraints of size proportional to $\Gamma\{x : \tau\} \vdash x : \tau$, which can be substantially larger.

3 Strictness inference system

We recall the strictness inference system of Amtoft, though adapted to the standard type inference system in Figure 1 instead of the one in Figure 2. Strictness properties are captured by refining the standard type system. We shall distinguish between a *strict* and a *(possibly) nonstrict* function type. The *strictness types* are then [Amt93]:

$$\sigma ::= \mathsf{Bool} \mid \sigma \rightarrow_0 \sigma \mid \sigma \rightarrow_1 \sigma$$

where $\sigma \rightarrow_0 \sigma'$ is the type of *strict* functions from σ to σ', and $\sigma \rightarrow_1 \sigma'$ the type of arbitrary — strict or nonstrict — functions from σ to σ'. This corresponds to *relevant* and *intuitionistic* implication, respectively, in [BF92]. We do not model *irrelevant* implication. The *strictness annotations* (subscripts) are ordered by $0 < 1$. The functional types induce a subtype relation as follows: $\sigma \leq \sigma$ and $\sigma \rightarrow_b \sigma' \leq \sigma'' \rightarrow_{b'} \sigma'''$ if $\sigma'' \leq \sigma, \sigma' \leq \sigma'''$ and $b \leq b'$. We denote strictness types by σ, the strictness indicators $0, 1$ by b, and the *erasure* of strictness annotations in σ or Γ by $|\sigma|$ and $|\Gamma|$, respectively. Note that strictness variables $b, b', \ldots$ and Boolean expressions over them are *metanotation* in the inference rules.

The strictness inference system is given in Figure 3. The rules are "annotated" versions of the static typing rules, with two additional rules (SUB1) and (SUB2) for subtyping. A strictness assumption is of the form $x^b : \sigma$. Strictness judgements consist of a sequence of strictness assumptions Γ and a conclusion of the form $e : \sigma$, written $\Gamma \vdash e : \sigma$ as usual. Note the role of the strictness annotations of variables in assumptions. Intuitively a 0-annotated variable in the assumptions is needed for any error-free terminating evaluation of the expression of the judgement. A 1-annotated variable may or may not be used in such an evaluation. A variable introduced in (WEAK) is labeled 1 since it is *not* needed in the evaluation of the expression at hand. If we have two assumptions for the same variable then the variable is needed if one of the assumptions carries a 0-label. Thus we can contract two such assumptions by conjuncting their labels. The subtyping rules express that a variable assumption $x^0 : \sigma$ (x is needed) can be weakened to $x^1 : \sigma$ (x is possibly not needed). The subtype relation on strictness types similarly weakens the strictness properties of the expression.

The variable annotation is transferred to the function type in the (ABSTR) rule: if the abstracted variable has annotation 0 then the λ-abstraction is strict. For $\Delta = x_1^{b_1} : \sigma_1, \ldots, x_n^{b_n} : \sigma_n$ the notation Δ^b denotes the assumptions $x_1^{b_1 \vee b} : \sigma_1, \ldots, x_n^{b_n \vee b} : \sigma_n$. This permits expressing the application of strict and nonstrict functions by the single rule (APPL). If $b = 0$ then the function expression e is strict and $\Delta^b = \Delta$. In this case the rule expresses that any variable that is needed in the evaluation of the argument e' to the function e is also needed in the evaluation of ee'. If $b = 1$, however, then e is possibly nonstrict. In this case $\Delta^b = x_1^1 : \sigma_1, \ldots, x_n^1 : \sigma_n$, which expresses that none of the variables needed in the evaluation of e' are sure to be needed in the evaluation of ee'.

4 Normalized strictness inference

Let us ignore applications of rule (EXCH) in derivations for now. For any closed expression e such that $\vdash e : \sigma$ there is a derivation of $\vdash e : \sigma$ such that:

- rule (SUB1) is only applied immediately after rule (TRIV);

$$\frac{\Gamma, x^b : \sigma, y^{b'} : \sigma', \Delta \vdash e : \sigma''}{\Gamma, y^{b'} : \sigma', x^b : \sigma, \Delta \vdash e : \sigma''} \quad \text{(EXCH)} \qquad \frac{\Gamma, x^b : \sigma, x^{b'} : \sigma, \Delta \vdash e : \sigma''}{\Gamma, x^{b \wedge b'} : \sigma, \Delta \vdash e : \sigma''} \quad \text{(CONTR)}$$

$$\frac{\Gamma, \Delta \vdash e : \sigma'}{\Gamma, x^1 : \sigma, \Delta \vdash e : \sigma'} \quad \text{(WEAK)} \qquad x^0 : \sigma \vdash x : \sigma \quad \text{(TRIV)}$$

$$\frac{\Gamma_x, x^b : \sigma \vdash e : \sigma'}{\Gamma_x \vdash \lambda x : |\sigma|.e : \sigma \to_b \sigma'} \quad \text{(ABSTR)} \qquad \frac{\Gamma \vdash e : \sigma \to_b \sigma' \qquad \Delta \vdash e' : \sigma}{\Gamma, \Delta^b \vdash ee' : \sigma'} \quad \text{(APPL)}$$

$$\frac{\Gamma_x, x^1 : \sigma \vdash e : \sigma}{\Gamma_x \vdash \mathbf{fix}\, x : |\sigma|.e : \sigma} \quad \text{(FIX)}$$

$$\vdash \mathit{true} : \mathsf{Bool} \quad (\text{CONST}_{\mathit{true}}) \qquad \vdash \mathit{false} : \mathsf{Bool} \quad (\text{CONST}_{\mathit{false}})$$

$$\frac{\Gamma \vdash e : \mathsf{Bool} \qquad \Delta \vdash e' : \sigma \qquad \Delta \vdash e'' : \sigma}{\Gamma, \Delta \vdash \mathbf{if}\, e\, e'\, e'' : \sigma} \quad \text{(COND)}$$

$$\frac{\Gamma, x^0 : \sigma, \Delta \vdash e : \sigma'}{\Gamma, x^1 : \sigma, \Delta \vdash e : \sigma'} \quad \text{(SUB1)} \qquad \frac{\Gamma \vdash e : \sigma \qquad (\sigma \leq \sigma')}{\Gamma \vdash e : \sigma'} \quad \text{(SUB2)}$$

Figure 3: Strictness type inference for functional language

- rule (SUB2) is only applied immediately after rule (TRIV), (SUB1), and (FIX);
- (WEAK) is only applied immediately before rule (ABSTR) if the variable introduced does not occur free in the expression of the judgement, or, possibly repeatedly, immediately before rule (COND);
- (CONTR) is always applied immediately and possibly repeatedly when there are two or more assumptions for the same variable in a judgement; no other rule is applied until there is at most one assumption per variable.

By building the application of rules (SUB1), (SUB2), (CONSTR), and (WEAK) into the nonstructural rules to which they can be attached we arrive at a syntax-directed inference system. We shall treat assumptions as *(multi)sets* instead of sequences. This dispenses with the need for rule (EXCH) altogether, and conversely this is sound due to the rule (EXCH). In other words, by using the more abstract sets instead of sequences we move rule (EXCH) to the metalevel.

The syntax-directed strictness inference system is given in Figure 4. It uses the following metanotation for assumptions. Let Γ, Δ be sets of assumptions

$$\begin{array}{rcl}\Gamma & = & \{x_1^{b_1} : \sigma_1, \dots, x_n^{b_n} : \sigma_n, y_1^{c_1} : \sigma'_1, \dots, y_m^{c_m} : \sigma'_m\} \\ \Delta & = & \{x_1^{b'_1} : \sigma''_1, \dots, x_n^{b'_n} : \sigma''_n, z_1^{d_1} : \sigma'''_1, \dots, z_k^{d_k} : \sigma'''_k\}\end{array}$$

where $\{x_1, \dots, x_n\}, \{y_1, \dots, y_m\}, \{z_1, \dots, z_k\}$ are pairwise disjoint sets of variables. We define $\Gamma \wedge \Delta$ and $\Gamma \vee \Delta$ by

$$\begin{array}{rcl}\Gamma \wedge \Delta & = & \{x_1^{b_1 \wedge b'_1} : \sigma_1, \dots, x_n^{b_n \wedge b'_n} : \sigma_n, y_1^{c_1} : \sigma'_1, \dots, y_m^{c_m} : \sigma'_m, z_1^{d_1} : \sigma'''_1, \dots, z_k^{d_k} : \sigma'''_k\} \\ \Gamma \vee \Delta & = & \{x_1^{b_1 \vee b'_1} : \sigma_1, \dots, x_n^{b_n \vee b'_n} : \sigma_n, y_1^{1} : \sigma'_1, \dots, y_m^{1} : \sigma'_m, z_1^{1} : \sigma'''_1, \dots, z_k^{1} : \sigma'''_k\}\end{array}$$

if $\sigma_1 = \sigma''_1, \dots, \sigma_n = \sigma''_n$ (otherwise $\Gamma \wedge \Delta$ and $\Gamma \vee \Delta$ are undefined). Recall that

$$\Gamma^b \quad = \quad \{x_1^{b_1 \vee b} : \sigma_1, \dots, x_n^{b_n \vee b} : \sigma_n, y_1^{c_1 \vee b} : \sigma'_1, \dots, y_m^{c_m \vee b} : \sigma'_m\}$$

By the considerations at the beginning of this section we obtain:

Proposition 1 $\vdash e : \sigma$ by Figure 3 if and only if $\vdash e : \sigma$ by Figure 4. □

5 Constraint characterization

Let $\vec{b}^+$ and $\vec{b}^-$ be sequences of strictness annotations 0 or 1. $()$ denotes the empty sequence. For a given standard type τ we let $\tau[\vec{b}^+, \vec{b}^-]$ be the strictness type we get by annotating the positive occurrences of function arrows in τ from left to right with $\vec{b}^+$ and the negative occurrences with $\vec{b}^-$ [Amt93]. (The result is undefined if the lengths of the sequences do not fit the type.) Formally,

$$\begin{array}{rcl}\mathsf{Bool}[(), ()] & = & \mathsf{Bool} \\ (\tau_1 \to \tau_2)[\vec{b}_1^+ b \vec{b}_2^+, \vec{b}_1^- \vec{b}_2^-] & = & \tau_1[\vec{b}_1^-, \vec{b}_1^+] \to_b \tau_2[\vec{b}_2^+, \vec{b}_2^-]\end{array}$$

Note that there is a unique way of writing every strictness type σ as $\tau[\vec{b}^+, \vec{b}^-]$. We extend the ordering $0 < 1$ on strictness annotations pointwise to sequences.

$$x^b : \sigma \vdash x : \sigma' \quad (\sigma \leq \sigma') \qquad \text{(TRIV')}$$

$$\frac{\Gamma_x, x^b : \sigma \vdash e : \sigma'}{\Gamma_x \vdash \lambda x : |\sigma|.e : \sigma \rightarrow_b \sigma'} \quad \text{(ABSTR')} \qquad \frac{\Gamma \vdash e : \sigma' \quad (x \text{ not free in } e)}{\Gamma \vdash \lambda x : |\sigma|.e : \sigma \rightarrow_1 \sigma'} \quad \text{(ABSTR")}$$

$$\frac{\Gamma \vdash e : \sigma \rightarrow_b \sigma' \qquad \Delta \vdash e' : \sigma}{\Gamma \wedge \Delta^b \vdash ee' : \sigma'} \quad \text{(APPL')}$$

$$\frac{\Gamma_x, x^b : \sigma \vdash e : \sigma \quad (\sigma \leq \sigma')}{\Gamma_x \vdash \textbf{fix}\, x : |\sigma|.e : \sigma'} \quad \text{(FIX')} \qquad \frac{\Gamma \vdash e : \sigma \quad (x \text{ not free in } e)}{\Gamma \vdash \textbf{fix}\, x : |\sigma|.e : \sigma} \quad \text{(FIX")}$$

$$\vdash \mathit{true} : \mathsf{Bool} \qquad (\text{CONST}_{\mathit{true}}) \qquad\qquad \vdash \mathit{false} : \mathsf{Bool} \qquad (\text{CONST}_{\mathit{false}})$$

$$\frac{\Gamma \vdash e : \mathsf{Bool} \qquad \Delta' \vdash e' : \sigma \qquad \Delta'' \vdash e'' : \sigma}{\Gamma \wedge (\Delta' \vee \Delta'') \vdash \textbf{if}\, e\, e'\, e'' : \sigma} \quad \text{(COND')}$$

Figure 4: Syntax-directed strictness inference system

$$x^b : \tau[\vec{b}_1^-, \vec{b}_1^+] \vdash x : \tau[\vec{b}_2^+, \vec{b}_2^-] \qquad (\vec{b}_2^+ \geq \vec{b}_1^-, \vec{b}_1^+ \geq \vec{b}_2^-)$$

$$\frac{\Gamma_x, x^{b^+} : \tau[\vec{b}^-, \vec{b}^+] \vdash e : \sigma'}{\Gamma_x \vdash \lambda x : \tau.e : \tau[\vec{b}^-, \vec{b}^+] \to_{b'^+} \sigma'} \quad (b'^+ = b^+)$$

$$\frac{\Gamma \vdash e : \sigma' \quad (x \text{ not free in } e)}{\Gamma \vdash \lambda x : \tau.e : \tau[\vec{b}^-, \vec{b}^+] \to_1 \sigma'}$$

$$\frac{\Gamma \vdash e : \tau[\vec{b}_1^-, \vec{b}_1^+] \to_b \sigma' \qquad \Delta \vdash e' : \tau[\vec{b}_2^+, \vec{b}_2^-]}{\Gamma' \vdash ee' : \sigma'} \quad \left(\begin{array}{l} \vec{b}_1^- = \vec{b}_2^+, \vec{b}_2^- = \vec{b}_1^+, \\ \Gamma' = \Gamma \wedge \Delta^b \end{array} \right)$$

$$\frac{\Gamma_x, x^b : \tau[\vec{b}_1^-, \vec{b}_1^+] \vdash e : \tau[\vec{b}_2^+, \vec{b}_2^-]}{\Gamma_x \vdash \textbf{fix}\, x : \tau.e : \tau[\vec{b}_3^+, \vec{b}_3^-]} \quad (\vec{b}_1^- = \vec{b}_2^+, \vec{b}_2^- = \vec{b}_1^+, \vec{b}_3^+ \geq \vec{b}_2^+, \vec{b}_2^- \geq \vec{b}_3^-)$$

$$\frac{\Gamma \vdash e : \tau[\vec{b}_2^+, \vec{b}_2^-] \quad (x \text{ not free in } e)}{\Gamma \vdash \textbf{fix}\, x : \tau.e : \tau[\vec{b}_2^+, \vec{b}_2^-]}$$

$$\vdash \mathit{true} : \mathsf{Bool} \qquad\qquad \vdash \mathit{false} : \mathsf{Bool}$$

$$\frac{\Gamma \vdash e : \mathsf{Bool} \qquad \begin{array}{l} \Delta' \vdash e' : \tau[\vec{b}'^+, \vec{b}'^-] \\ \Delta'' \vdash e'' : \tau[\vec{b}''^+, \vec{b}''^-] \end{array}}{\Gamma' \vdash \textbf{if}\, e\, e'\, e'' : \tau[\vec{b}'''^+, \vec{b}'''^-]} \quad \left(\begin{array}{l} \vec{b}'''^+ = \vec{b}'^+, \vec{b}'^- = \vec{b}'''^-, \\ \vec{b}'''^+ = \vec{b}''^+, \vec{b}''^- = \vec{b}'''^- \\ \Gamma' = \Gamma \wedge (\Delta' \vee \Delta'') \end{array} \right)$$

Figure 5: Constraint-based strictness type inference for functional language

Proposition 2 $\tau[\vec{b}_1^+, \vec{b}_1^-] \leq \tau'[\vec{b}_2^+, \vec{b}_2^-]$ if and only if $\tau = \tau'$ and $\vec{b}_1^+ \leq \vec{b}_2^+, \vec{b}_2^- \leq b_1^-$. □

Using this notation we present another strictness inference system in Figure 5. We use explicit side conditions on strictness annotations in order to stage the transition to constructing a constraint system that characterizes *all* derivable strictness properties of a (closed) program expression. The side conditions on assumption sets in the rules for application and conditional can be decomposed into the definition of the assumption set in the consequent of the rule and side conditions on the strictness annotations occurring in the assumptions.

In the rule for application, if

$$\begin{array}{rcl}
\Gamma & = & \{x_1^{b_1} : \tau_1[\vec{b}_1^-, \vec{b}_1^+], \ldots, x_n^{b_n} : \tau_n[\vec{b}_n^-, \vec{b}_n^+], y_1^{c_1} : \tau_1'[\vec{c}_1^-, \vec{c}_1^+], \ldots, y_m^{c_m} : \tau_m'[\vec{c}_m^-, \vec{c}_m^+]\} \\
\Delta & = & \{x_1^{b_1'} : \tau_1[\vec{b}_1'^-, \vec{b}_1'^+], \ldots, x_n^{b_n'} : \tau_n[\vec{b}_n'^-, \vec{b}_n'^+], z_1^{d_1} : \tau_1''[\vec{d}_1^-, \vec{d}_1^+], \ldots, z_k^{d_k} : \tau_k''[\vec{d}_k^-, \vec{d}_k^+]\}
\end{array}$$

with $\{x_1, \ldots, x_n\}, \{y_1, \ldots, y_m\}, \{z_1, \ldots, z_k\}$ pairwise disjoint then the assumption set Γ' in the conclusion of the rule is

$$\begin{array}{rcl}
\Gamma' & = & \{x_1^{b_1''} : \tau_1[\vec{b}_1''^-, \vec{b}_1''^+], \ldots, x_n^{b_n''} : \tau_n[\vec{b}_n''^-, \vec{b}_n''^+], \\
& & y_1^{c_1} : \tau_1'[\vec{c}_1^-, \vec{c}_1^+], \ldots, y_m^{c_m} : \tau_m'[\vec{c}_m^-, \vec{c}_m^+], \\
& & z_1^{d_1'} : \tau_1''[\vec{d}_1^-, \vec{d}_1^+], \ldots, z_k^{d_k'} : \tau_k''[\vec{d}_k^-, \vec{d}_k^+]\}
\end{array}$$

and the side conditions $\Gamma' = \Gamma \wedge \Delta^b$ on strictness annotations occurring in Γ, Δ, Γ' are

$$\left(\begin{array}{l}
\vec{b}_1''^+ = \vec{b}_1^+, \vec{b}_1^- = \vec{b}_1''^-, \vec{b}_1''^+ = \vec{b}_1'^+, \vec{b}_1'^- = \vec{b}_1''^-, \ldots, \\
\vec{b}_n''^+ = \vec{b}_n^+, \vec{b}_n^- = \vec{b}_n''^-, \vec{b}_n^+ = \vec{b}_n'^+, \vec{b}_n'^- = \vec{b}_n^-, \\
b_1'' = b_1 \wedge (b_1' \vee b), \ \ldots \ , b_n'' = b_n \wedge (b_n' \vee b), \\
d_1' = d_1 \vee b, \ \ldots \ , d_k' = d_k \vee b.
\end{array}\right)$$

Similarly for conditionals.

The side conditions are formulated carefully such that an equational condition $b = s$ can be replaced by the inequational condition $b \geq s$. This "loosening" in the equational condition corresponds to insertion of subtyping rules (SUB1) and/or (SUB2). Conversely, Proposition 1 says that for any $\vdash e : \sigma$ there exists a derivation that satisfies such a loosened side condition equationally.

Every standard type derivation induces canonically a *schematic* strictness derivation with:

- *uninstantiated* strictness (meta)variables $b_1, b_2, \ldots$ that occur *at most once* in any judgement in the derivation, and
- *(formal) constraints* $b = s$ or $b \geq s$ denoting the side conditions for all the rule instances used in the derivation.

Here b ranges over strictness variables and s over Boolean expressions constructed from such variables, $\wedge, \vee$. We write $\Gamma \vdash e : \sigma[C]$ if $\Gamma \vdash e : \sigma$ is the root of this schematic derivation, and C is the union of all the formal constraints occurring in the derivation.

A *solution* S of C is a mapping of the strictness variables to $\mathcal{B} = \{0, 1\}$ such that all the constraints in C *hold* (are satisfied). Every solution of C gives a valid strictness judgement; that is, we have $S(\Gamma) \vdash e : S(\sigma)$, and vice versa. Even stronger, if $\Delta \vdash e : \sigma'$ (by Figure 4) and $|\Delta| = |\Gamma|, |\sigma| = |\sigma'|$ then $\Delta = S(\Gamma)$ and $\sigma' = S(\sigma)$ for a suitable solution S of C.

Lemma 3 If $\Gamma \vdash e : \sigma[C]$ then:

- the strictness variables in negative position in $\Gamma \vdash e : \sigma$ occur only on the right-hand sides of constraints in C;
- the strictness variables in positive position in $\Gamma \vdash e : \sigma$ occur only on the left-hand sides of constraints in C.

Variables occurring in C, but not in $\Gamma \vdash e : \sigma$ may occur on either or both sides of constraints in C. □

Proof By induction on derivations in Figure 5. □

Note that we consider annotation b in assumption $x^b : \sigma$ to be *positive* whereas the polarity of annotations in σ is *flipped*.

The constraints in C are either equational ($b = s$) or inequational ($b \geq s$). Recall that an equational constraint $b = s$ can be replaced with the inequational constraint $b \geq s$. Treating the constraint system as a set of inequalities we can transform them to a form where every variable occurs at most once on the left-hand side of some constraint: $b \geq s, b \geq s'$ is transformed to $b \geq s \vee s'$. From fixed point theory we get that the least solution of these constraints exists, is unique and solves all of the constraints equationally, given *any* asignment of strictness values to the negative strictness variables:

Theorem 4 For given assignment of values to strictness variables occurring in negative position (only) there is a *unique least* assignment of values to strictness variables in positive position (only) such that $\Gamma \vdash e : \sigma$ is derivable. □

6 Complexity of strictness type inference

The constraints C in a schematic derivation $\Gamma \vdash e : \sigma[C]$ are efficiently solvable by an iterative fixed point algorithm.

Theorem 5 Let m be the number of distinct strictness variables in C. Then C is of size $O(m)$ and for any (initial) values for the strictness variables occurring in negative position the constraint system can be solved for the least solution in time $O(m)$. □

Proof There is a constant k such that every constraint is of size at most k. Every variable occurs at most once on the left-hand side of a constraint. Thus the total number of constraints is $O(m)$. Since they are each of constant size the total size of the constraints is also $O(m)$.

The constraint system can be solved in time linear in its size by standard iterative fixed point algorithms; see *e.g.* [Kil73,Tar76]. □

A practically efficient implementation would eliminate all equational constraints between variables symbolically first (by unification), and solve the remaining constraints iteratively, treating all of them as inequalities. The number of remaining constraints will typically be substantially smaller than the number of original constraints. (See [CP89] for an analysis and examples of hybrid elimination/iteration fixed point methods.)

Given an *explicitly* simply typed program expression e of size n (measured in the number of symbols in e) the value of m is $\Theta(n^2)$. The lower bound arises from

$$e_n = \lambda x : \mathsf{Int}^{n+1}.\lambda y : \mathsf{Int}^n.(\underbrace{xy, \ldots, xy}_{n})$$

where $\tau^n = \underbrace{\tau \to \ldots \to \tau}_{n}$. The upper bound can be checked by "charging" a strictness variable to either a standard type occurring in the expression or to a symbol in the (untyped part of the) program expression. The worst case arises when the number of occurrences of a *single* variable is on the same order of magnitude as the size of all explicit type occurrences in the program expression. If we bound the maximum number of occurrences of any single variable by k, though, then $m = O(kn)$. Thus we would expect that the constraint system generated for an explicitly typed program expression occurring in practice is proportional in size to the program expression itself.

Incidentally, the same bound of $\Theta(n^2)$ also appears to hold for the size of the constraint system generated by Amtoft. However, if we bound the number of occurrences of any single variable to k then the bound is still $\Theta(n^2)$ since the bound arises as the product of the size of all occurrences of types in an expression and the *total* number of variables occurrences (and not just of a *single* variable). This is due to the fact that the number of constraints generated for a variable occurrence in Amtoft's constraint formulation is proportional to the number of open variable scopes (λ- or **fix** -binding operators) within which the variable occurs.

Note that the requirement that the program expression be explicitly typed is essential as n may be exponentially bigger than the underlying implicitly typed program expression. For example,

$$e_k = \lambda x_1 \ldots x_k . x_1(x_1 x_2)(x_2 x_3) \ldots (x_{k-1} x_k)$$

is of size $\Theta(k)$ as an implicitly typed expression, but of size $\Theta(2^k)$ as an explicitly typed expression.[2]

For typical programs arising in practice we should expect that their explicitly typed version is of the same order of size as the implicitly typed version. Thus our strictness analysis algorithm should work satisfactorily even for large programs.

In the worst case, however, given an *untyped* program expression of size l constructing and solving a constraint system for it can be as bad as $\Theta(2^{cl})$ in the worst case. It is conceivable that a compact representation of strictness information — analogous to the well-known dag-representation for simple types — could lead to substantially better performance, both asymptotically and in practical use.

References

[Amt93] Torben Amtoft. Strictness types: An inference algorithm and an application. Technical Report PB-448, DAIMI, Aarhus Universit, Ny Munkegade, Building 540, DK-8000 Aarhus C, Denmark, Aug. 1993.

[BF92] Clement Baker-Finch. Relevant logic and strictness analysis. In *Proc. Workshop on Static Analysis (WSA), Bordeaux, France*, pages 221–228, Sept. 1992.

[CP89] J. Cai and R. Paige. Program derivation by fixed point computation. *Science of Computer Programming*, 11:197–261, 1989.

[Kil73] G. Kildall. A unified approach to global program optimization. *Proc. ACM Symp. on Principles of Programming Languages (POPL)*, 1973.

[2]Just enter e_k for $k > 16$ or so into your favorite typed functional programming language processor to obtain the type of e_k.

[Tar76] R. Tarjan. Iterative algorithms for global flow analysis. In J. Traub, editor, *Algorithms and Complexity*, pages 91–102. Academic Press, 1976.

[Wri91] D. Wright. A new technique for strictness analysis. In *Proc. Int'l J. Conf. on Theory and Practice of Software Development (TAPSOFT)*, April 1991.