Minimal Thunkification

Torben Amtoft [⋆] [⋆⋆]
internet: `tamtoft@daimi.aau.dk`

Computer Science Department, Aarhus University
Ny Munkegade, DK-8000 Århus C, Denmark

Abstract. By "thunkifying" the arguments to function applications and "dethunkifying" variables one can translate a λ-expression e into a λ-expression e', such that call-by-value evaluation of e' gives the same result as call-by-name evaluation of e. By using the result of a strictness analysis, some of these thunkifications can be avoided. In this paper we present a type system for strictness analysis; present a translation algorithm which exploits the strictness proof tree; and give a combined proof of the correctness of the analysis/translation.

1 Introduction

We shall consider the following problem: given λ-expression e, find a λ-expression e' such that e when evaluated using a call-by-name strategy yields the same result as e' when evaluated using a call-by-value strategy. The reason why this is interesting is that it is more convenient [Hug89] to program in a lazy language than in an eager (the former also enjoys the nice property of referential transparency); and that CBV traditionally is considered more efficient than CBN.

The standard technique to solve the problem is to introduce "thunks" everywhere (thus simulating how one would naively implement call-by-name), as done e.g. in [DH92]. That is, we have the following translation T:

- An abstraction $\lambda x.e$ translates into $\lambda x.T(e)$;
- An application $e_1 e_2$ translates into $T(e_1)(\lambda x.T(e_2))$ (where x is a fresh variable) – that is, the evaluation of the argument is suspended ("thunkified");
- A variable x translates into $(x\ d)$ (where d is a "dummy" argument) – since x will become bound to a suspension x must be "dethunkified".

Clearly, this is far from optimal since many expressions may become thunkified only to become dethunkified soon after. This kind of observation motivated Mycroft [Myc80] to introduce *strictness analysis* by means of abstract interpretation: if $e_1^{\#}$ is the denotation of e_1 in the abstract domain, and if $e_1^{\#}(\bot) = \bot$, then it is safe to omit thunkification of the argument to e_1. A thunkification algorithm exploiting strictness information is presented in [DH93].

⋆ Supported by the DART-project funded by the Danish Research Council.
⋆⋆ The work reported here evolved from numerous discussions with Hanne Riis Nielson and Flemming Nielson. Also thanks to Jens Palsberg for useful feedback.

Strictness analysis for higher-order functions is treated in [BHA86] and proved correct in the following sense: if $e_1^\#$ is the abstract denotation of e_1, and if $e_1^\#(\perp) =\perp$ (in the abstract domain), then $e_1(\perp) =\perp$ (in the concrete domain) – that is, e_1 will not terminate if its argument does not terminate. However, no attention is given to proving the correctness of a translation using this information.

This is a quite general phenomenon, cf. the claims made in [Wan93]:

> The goal of flow analysis is to annotate a program with certain propositions about the behavior of that program. One can then apply optimizations to the program that are justified by those propositions. However, it has proven remarkably difficult to specify the semantics of those propositions in a way that justifies the resulting optimizations.

The main contribution of this paper is to give a combined proof of the correctness of a strictness analysis and of the resulting transformation. This can be seen as following the trend of [Wan93] who proves the combined correctness of a binding time analysis and a partial evaluation based on the result of this analysis. Also something similar can be found in [Lan92] where the correctness of a code generation exploiting strictness information is proved.

The strictness analysis to be used in this paper will be formulated in terms of inference rules for a type system, where functions arrows have been annotated: $\rightarrow_0$ denoting that a function is strict and $\rightarrow_1$ denoting that we do not know anything for sure. This is inspired by the method of Wright [Wri91] – in [Wri92] he proves the correctness of his *analysis* (by means of a model for the λ-calculus), but does not consider any transformation based on the result of the analysis.

Other type-based approaches to strictness analysis includes [KM89], where the base types (not the arrows) are annotated with strictness information. An attempt to clarify the relation between strictness analysis based on abstract interpretation resp. type inference is presented in [Jen91] – however, the relationship is in no way fully understood. Neither is the relative power of the various approaches in the literature, and accordingly the strength of our analysis will not be compared formally with other analyses.

An Overview of the Paper

In Sect. 2 we present the syntax and semantics of the language to be considered, in particular we give an inference system for assigning (ordinary) types to expressions. In Sect. 3 we present an inference system for *strictness* types, such that any expression which can be given an (ordinary) type also can be given a strictness type. Section 4 gives a translation algorithm (which exploits the proof tree generated by the strictness analysis). In Sect. 5 we formulate predicates expressing the correctness of the translation/analysis, and we briefly outline a correctness proof – for a full proof, see [Amt93].

2 The λ-Calculus with Constants

Expressions. An expression is either a variable x; a constant c; an abstraction $\lambda x.e$; an application $e_1 e_2$; an unbounded recursive definition $\mathsf{rec}\ f\ e$; a bounded recursive definition $\mathsf{rec}_n\ f\ e$ (with $n \geq 0$) or a conditional $\mathsf{if}\ e_1\ e_2\ e_3$. The set of free variables in e will be denoted $\mathsf{FV}(e)$.

Of course, the user will only write programs with unbounded recursion – bounded recursion is introduced as an auxiliary device for proving the correctness of the translation (cf. Theorem 6 and Theorem 7).

The reason for not making if a constant (thereby making it possible to dispense with the conditional) is that if is a *non-strict* constant and hence requires special treatment.

Types. The set of (ordinary) types will be denoted $\mathcal{T}$; such a type is either a base type (Int, Bool, Unit etc.) or a function type $t_1 \to t_2$. Base will denote some base type.

An *iterated base type* is either Base or of form $\mathsf{Base} \to t$, where t is an iterated base type. We shall assume that there exists a function Ct which assigns iterated base types to all constants.

In Fig. 1 we present a type inference system, where inferences are of form $\Gamma \vdash e : t$. Here Γ is an environment assigning types to (a superset of) the free variables of e. For closed expressions q it makes sense to say that q is of type t, since if $\Gamma \vdash q : t$ then also $\Gamma' \vdash q : t$ for any Γ'.

$$\Gamma \vdash c : Ct(c) \qquad\qquad \Gamma \vdash x : \Gamma(x)$$

$$\frac{\Gamma \cup \{x := t_1\} \vdash e : t}{\Gamma \vdash \lambda x.e : t_1 \to t} \qquad\qquad \frac{\Gamma \vdash e_1 : t_2 \to t_1,\ \Gamma \vdash e_2 : t_2}{\Gamma \vdash e_1 e_2 : t_1}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{Bool},\ \Gamma \vdash e_2 : t,\ \Gamma \vdash e_3 : t}{\Gamma \vdash (\mathsf{if}\ e_1\ e_2\ e_3) : t}$$

$$\frac{\Gamma \cup \{f := t\} \vdash e : t}{\Gamma \vdash (\mathsf{rec}\ f\ e) : t} \qquad\qquad \frac{\Gamma \cup \{f := t\} \vdash e : t}{\Gamma \vdash (\mathsf{rec}_n\ f\ e) : t}$$

Fig. 1. An inference system for (ordinary) types.

Semantics. We say that an expression is in weak head normal form (WHNF) if it is either a constant c or of form $\lambda x.e$. As no constructors are present in the language, this choice of normal form will be suitable for CBV as well as for CBN.

We define a SOS for call-by-name (Fig. 2) and a SOS for call-by-value (Fig. 3), with inferences of form $q \Rightarrow_N q'$ resp. $q \Rightarrow_V q'$. Here q and q' are *closed* expressions. We assume the existence of a function $\mathsf{Applycon}$ such that for two constants c_1 and c_2, $\mathsf{Applycon}(c_1, c_2)$ *either* yields another constant c such that if $Ct(c_1) = \mathsf{Base} \rightarrow t$, $Ct(c_2) = \mathsf{Base}$ then $Ct(c) = t$ *or* the expression $c_1 c_2$ itself (to model errors). For instance, $\mathsf{Applycon}(+, 4)$ could be the constant $+_4$, where $\mathsf{Applycon}(+_4, 3)$ is the constant 7. To model that division by zero is illegal we let e.g. $\mathsf{Applycon}(/_7, 0) = (/_7, 0)$.

$$
\begin{array}{ll}
(\lambda x.e)q \Rightarrow_N e[q/x] & \dfrac{q_1 \Rightarrow_N q_1'}{q_1 q_2 \Rightarrow_N q_1' q_2} \\[2ex]
c_1 c_2 \Rightarrow_N \mathsf{Applycon}(c_1, c_2) & \dfrac{q_2 \Rightarrow_N q_2'}{c q_2 \Rightarrow_N c q_2'} \\[2ex]
\mathsf{if\ True}\ q_2\ q_3 \Rightarrow_N q_2 & \mathsf{if\ False}\ q_2\ q_3 \Rightarrow_N q_3 \\[2ex]
\dfrac{q_1 \Rightarrow_N q_1'}{\mathsf{if}\ q_1\ q_2\ q_3 \Rightarrow_N \mathsf{if}\ q_1'\ q_2\ q_3} & \mathsf{rec}\ f\ e \Rightarrow_N e[(\mathsf{rec}\ f\ e)/f] \\[2ex]
\mathsf{rec}_0\ f\ e \Rightarrow_N \mathsf{rec}_0\ f\ e & \mathsf{rec}_{n+1}\ f\ e \Rightarrow_N e[(\mathsf{rec}_n\ f\ e)/f]
\end{array}
$$

Fig. 2. A SOS for CBN.

$$
\begin{array}{ll}
(\lambda x.e)q \Rightarrow_V e[q/x],\ \text{if } q \text{ in } \mathsf{WHNF} & \dfrac{q_1 \Rightarrow_V q_1'}{q_1 q_2 \Rightarrow_V q_1' q_2} \\[2ex]
c_1 c_2 \Rightarrow_V \mathsf{Applycon}(c_1, c_2) & \dfrac{q_2 \Rightarrow_V q_2'}{q_1 q_2 \Rightarrow_V q_1 q_2'},\ \text{if } q_1 \text{ in } \mathsf{WHNF} \\[2ex]
\mathsf{if\ True}\ q_2\ q_3 \Rightarrow_V q_2 & \mathsf{if\ False}\ q_2\ q_3 \Rightarrow_V q_3 \\[2ex]
\dfrac{q_1 \Rightarrow_V q_1'}{\mathsf{if}\ q_1\ q_2\ q_3 \Rightarrow_V \mathsf{if}\ q_1'\ q_2\ q_3} & \mathsf{rec}\ f\ e \Rightarrow_V e[(\mathsf{rec}\ f\ e)/f] \\[2ex]
\mathsf{rec}_0\ f\ e \Rightarrow_V \mathsf{rec}_0\ f\ e & \mathsf{rec}_{n+1}\ f\ e \Rightarrow_V e[(\mathsf{rec}_n\ f\ e)/f]
\end{array}
$$

Fig. 3. A SOS for CBV.

We have the following (standard) result (which exploits that all constants are of iterated base type, as otherwise $c(\lambda x.e)$ might be well-typed but stuck – we also need the extra assumption that if $Ct(c) = \mathsf{Bool}$ then $c = \mathsf{True}$ or $c = \mathsf{False}$).

Fact 1. *Suppose (with q closed) $\Gamma \vdash q : t$. Then either q is in* WHNF*, or there exists unique q' such that $q \Rightarrow_N q'$ and such that $\Gamma \vdash q' : t$.*
Similarly for $\Rightarrow_V$.

We will introduce a "canonical" looping term Ω, defined by $\Omega = \mathsf{rec}\ f\ f$. There exists no q in WHNF such that $\Omega \Rightarrow_N^* q$ (or $\Omega \Rightarrow_V^* q$), but for all types t (and all Γ) we have $\Gamma \vdash \Omega : t$.

Thunkification and Dethunkification. We shall use the following notation: if t is a type in $\mathcal{T}$, $[t]$ is a shorthand for $\mathsf{Unit} \rightarrow t$.

If e is an expression, let $\underline{e}$ be a shorthand for $\lambda x.e$, where x is a fresh variable.

If e is an expression, let $\mathcal{D}(e)$ be a shorthand for $e\ d$, where d is a dummy constant of type Unit.

Fact 2. *If $\Gamma \vdash e : t$, then $\Gamma \vdash \underline{e} : [t]$. If $\Gamma \vdash e : [t]$, then $\Gamma \vdash \mathcal{D}(e) : t$.*
For all e, $\mathcal{D}(\underline{e}) \Rightarrow_N e$ and $\mathcal{D}(\underline{e}) \Rightarrow_V e$.

3 Strictness Types

The set of strictness types, $\mathcal{T}_{\mathrm{sa}}$, is defined as follows: a strictness type t is either a base type Base or a *strict* function type $t_1 \rightarrow_0 t_2$ (denoting that we *know* that the function is strict) or a *general* function type $t_1 \rightarrow_1 t_2$ (denoting that we do not know whether the function is strict).

We shall impose an ordering $\leq$ on strictness types, defined by stipulating that $t_1 \rightarrow_b t_2 \leq t_1' \rightarrow_{b'} t_2'$ iff $t_1' \leq t_1$, $b \leq b'$ and $t_2 \leq t_2'$, and by stipulating that $\mathsf{Int} \leq \mathsf{Int}$ etc. $t \leq t'$ means that t is more informative than t'; for instance it is more informative to know that a function is of type $\mathsf{Int} \rightarrow_0 \mathsf{Int}$ than to know that it is of type $\mathsf{Int} \rightarrow_1 \mathsf{Int}$.

We define two kinds of mappings from $\mathcal{T}_{\mathrm{sa}}$ into $\mathcal{T}$, E and Z, with the following intended meaning: if e can be assigned strictness type t and (the CBN-term) e translates into an equivalent CBV-term e', then e has type $\mathsf{E}(t)$ and e' has type $Z(t)$. E simply removes annotations from arrows, while Z in addition thunkifies arguments to non-strict functions. That is, we have

- $\mathsf{E}(\mathsf{Base}) = \mathsf{Base}$, $\mathsf{E}(t_1 \rightarrow_0 t_2) = \mathsf{E}(t_1) \rightarrow \mathsf{E}(t_2)$, $\mathsf{E}(t_1 \rightarrow_1 t_2) = \mathsf{E}(t_1) \rightarrow \mathsf{E}(t_2)$.
- $Z(\mathsf{Base}) = \mathsf{Base}$, $Z(t_1 \rightarrow_0 t_2) = Z(t_1) \rightarrow Z(t_2)$, $Z(t_1 \rightarrow_1 t_2) = [Z(t_1)] \rightarrow Z(t_2)$.

A *strict iterated base type* is either Base or of form $\mathsf{Base} \rightarrow_0 t$, where t is a strict iterated base type. Ct can be uniquely extended into CT_{sa}, a mapping from constants to strictness types, by demanding that with $t = CT_{\mathrm{sa}}(c)$, t must be a strict iterated base type with $\mathsf{E}(t) = Ct(c)$ (recall that the non-strict constant if has been given a special status).

In Fig. 4 we present an inference system for strictness types. An inference is now of the form $\Gamma, T \vdash_{\mathrm{sa}} e : t, W$. Here

- Γ is an environment assigning strictness types to variables;
- e is an expression such that if $x \in \mathsf{FV}(e)$ then $\Gamma(x)$ is defined;
- t is a strictness type;
- W is a subset of $\mathsf{FV}(e)$. It might be helpful to think of W as variables which are needed in order to evaluate e to "head normal form".
- T is a subset of the domain of Γ, denoting those variables which have been bound by non-strict λ-abstractions in the given context (T is used for recording purposes only).

The first inference rule is non-structural and expresses the ability to forget information: if an expression has type t and needs the variables in W, it also has a more imprecise type and will also need a subset of W. The application of this rule might for instance be needed in order to assign the same type to the two branches in a conditional. The two rules for abstractions (among other things) say that if x is among the variables needed by e then $\lambda x.e$ can be assigned a strict type ($\rightarrow_0$), otherwise not. The two rules for applications (among other things) say that if e_1 is strict then the variables needed by e_2 will also be needed by $e_1 e_2$.

We have $\Gamma, T \vdash_{\mathrm{sa}} \Omega : t, \emptyset$ for all strictness types t. An expression which can be assigned a strictness type can also be assigned an ordinary type:

Fact 3. *Suppose $\Gamma, T \vdash_{sa} e : t, W$. Then $\mathsf{E}(\Gamma) \vdash e : \mathsf{E}(t)$.*

Conversely, an expression which can be assigned an ordinary type can also be assigned at least one strictness type:

Fact 4. *Suppose $\Gamma \vdash e : t$. Suppose Γ' is such that $\mathsf{E}(\Gamma') = \Gamma$, and such that for all x all arrows in $\Gamma'(x)$ are annotated 1. Then (for all T) there exists t' and W, with $\mathsf{E}(t') = t$, such that $\Gamma', T \vdash_{sa} e : t', W$.*

Proof. An easy induction; choose $W = \emptyset$ and t' as t with all $\rightarrow$'s replaced by $\rightarrow_1$. We use that $CT_{\mathrm{sa}}(c)$ is a strict iterated base type and hence is least among all types t with $\mathsf{E}(t) = \mathsf{E}(CT_{\mathrm{sa}}(c))$. $\square$

The type system is rather similar to the one of Wright [Wri91] where function arrows are marked by boolean expressions – a major difference is that he imposes a "substitution ordering" (which hence is monotone in both "arrow positions") among types.

Wrt. other approaches, the following two examples will briefly hint at the relative strength of our type system – recall that the main point of this paper is *not* to present a superior strictness analysis!

Example 1. Consider the function f defined by $\mathsf{rec}\ f\ \lambda x.\lambda y.\lambda z.e$ where $e = \mathsf{if}\ (z = 0)\ (x + y)\ (f\ y\ x\ (z - 1))$. f is strict in all its arguments, but this cannot be inferred by the type system from [KM89] (due to the lack of conjunction types). In our system, however, we have

$$\emptyset, \emptyset \vdash_{\mathrm{sa}} \mathsf{rec}\ f\ \lambda x.\lambda y.\lambda z.e : \mathsf{Int} \rightarrow_0 \mathsf{Int} \rightarrow_0 \mathsf{Int} \rightarrow_0 \mathsf{Int}, \emptyset$$

$$\frac{\Gamma, T \vdash_{\mathrm{sa}} e : t, W}{\Gamma, T \vdash_{\mathrm{sa}} e : t', W'} \text{ if } t \leq t', W' \subseteq W$$

$$\Gamma, T \vdash_{\mathrm{sa}} c : CT_{\mathrm{sa}}(c), \emptyset$$

$$\Gamma, T \vdash_{\mathrm{sa}} x : \Gamma(x), \{x\}$$

$$\frac{\Gamma \cup \{x := t_1\}, T \vdash_{\mathrm{sa}} e : t, W \cup \{x\}}{\Gamma, T \vdash_{\mathrm{sa}} \lambda x.e : t_1 \rightarrow_0 t, W} \text{ if } x \notin T$$

$$\frac{\Gamma \cup \{x := t_1\}, T \cup \{x\} \vdash_{\mathrm{sa}} e : t, W}{\Gamma, T \vdash_{\mathrm{sa}} \lambda x.e : t_1 \rightarrow_1 t, W} \text{ if } x \notin W$$

$$\frac{\Gamma, T \vdash_{\mathrm{sa}} e_1 : t_2 \rightarrow_0 t_1, W_1 \ \ \Gamma, T \vdash_{\mathrm{sa}} e_2 : t_2, W_2}{\Gamma, T \vdash_{\mathrm{sa}} e_1 e_2 : t_1, W_1 \cup W_2}$$

$$\frac{\Gamma, T \vdash_{\mathrm{sa}} e_1 : t_2 \rightarrow_1 t_1, W_1 \ \ \Gamma, T \vdash_{\mathrm{sa}} e_2 : t_2, W_2}{\Gamma, T \vdash_{\mathrm{sa}} e_1 e_2 : t_1, W_1}$$

$$\frac{\Gamma, T \vdash_{\mathrm{sa}} e_1 : \mathsf{Bool}, W_1 \ \ \Gamma, T \vdash_{\mathrm{sa}} e_2 : t, W_2 \ \ \Gamma, T \vdash_{\mathrm{sa}} e_3 : t, W_3}{\Gamma, T \vdash_{\mathrm{sa}} (\mathsf{if}\ e_1\ e_2\ e_3) : t, W_1 \cup (W_2 \cap W_3)}$$

$$\frac{\Gamma \cup \{f := t\}, T \vdash_{\mathrm{sa}} e : t, W}{\Gamma, T \vdash_{\mathrm{sa}} (\mathsf{rec}\ f\ e) : t, W} \text{ if } f \notin W, f \notin T$$

$$\frac{\Gamma \cup \{f := t\}, T \vdash_{\mathrm{sa}} e : t, W}{\Gamma, T \vdash_{\mathrm{sa}} (\mathsf{rec}_n\ f\ e) : t, W} \text{ if } f \notin W, f \notin T$$

Fig. 4. An inference system for strictness types.

This is because we – with $\Gamma_1 = \{f := \mathsf{Int} \rightarrow_0 \mathsf{Int} \rightarrow_0 \mathsf{Int} \rightarrow_0 \mathsf{Int}\}$ – have

$$\Gamma_1, \emptyset \vdash_{\mathrm{sa}} \lambda x.\lambda y.\lambda z.e : \mathsf{Int} \rightarrow_0 \mathsf{Int} \rightarrow_0 \mathsf{Int} \rightarrow_0 \mathsf{Int}, \emptyset$$

which again is because we – with $\Gamma_2 = \Gamma_1 \cup \{x := \mathsf{Int}, y := \mathsf{Int}, z := \mathsf{Int}\}$ – have

$$\Gamma_2, \emptyset \vdash_{\mathrm{sa}} \mathsf{if}\ (z = 0)\ (x + y)\ (f\ y\ x\ (z - 1)) : \mathsf{Int}, \{x, y, z\}$$

This follows from the fact that $\Gamma_2, \emptyset \vdash_{\mathrm{sa}} (z = 0) : \mathsf{Bool}, \{z\}$ and $\Gamma_2, \emptyset \vdash_{\mathrm{sa}} (x + y) : \mathsf{Int}, \{x, y\}$ and

$$\Gamma_2, \emptyset \vdash_{\mathrm{sa}} (f\ y\ x\ (z - 1)) : \mathsf{Int}, \{x, y, z\}$$

The latter follows since e.g. $\Gamma_2, \emptyset \vdash_{\mathrm{sa}} (f\ y) : \mathsf{Int} \rightarrow_0 \mathsf{Int} \rightarrow_0 \mathsf{Int}, \{y\}$.

Example 2. Our analysis is not very good at handling recursive definitions with free variables. To see this, consider the function g given by

$$\lambda y.\mathsf{rec}\ f\ \lambda x.\mathsf{if}\ (x = 0)\ y\ (f\ (x - 1))$$

Clearly ge_1e_2 will loop if e_1 loops, so the analysis *ought* to conclude that g has strictness type $\mathsf{Int}{\rightarrow_0}\mathsf{Int}{\rightarrow_0}\mathsf{Int}$. However, we can do no better than inferring that g has strictness type $\mathsf{Int}{\rightarrow_1}\mathsf{Int}{\rightarrow_0}\mathsf{Int}$ – this is because it is impossible to deduce $\ldots\vdash_{\mathrm{sa}}(\mathsf{rec}\ f\ \ldots) : \ldots, \{y\}$ which in turn is because it is impossible to deduce $\ldots\vdash_{\mathrm{sa}}(\mathsf{if}\ (x = 0)\ y\ (f\ (x-1))) : \ldots, \{x, y\}$. The reason for this is that we cannot record in $\Gamma(f)$ that f needs y.

In order to repair on that, function arrows should be annotated not only with 0/1 but also with which free variables are needed – at the cost of complicating the theory significantly.

Inferring Strictness Types

First notice that no "least typing property" holds: the expression $\lambda f.f$ has type $(\mathsf{Int}{\rightarrow_0}\mathsf{Int}){\rightarrow_0}(\mathsf{Int}{\rightarrow_0}\mathsf{Int})$ and type $(\mathsf{Int}{\rightarrow_1}\mathsf{Int}){\rightarrow_0}(\mathsf{Int}{\rightarrow_1}\mathsf{Int})$ but *not* type $(\mathsf{Int}{\rightarrow_1}\mathsf{Int}){\rightarrow_0}(\mathsf{Int}{\rightarrow_0}\mathsf{Int})$.

On the other hand, it is possible to develop a type inference algorithm which for each assignment to the arrows occurring in *contravariant* position finds a *least* assignment to the arrows in *covariant* position. The algorithm works by solving constraints "on the fly" and is fully described in [Amt93]; below we shall give a brief outline:

The first step is to reformulate the inference system from Fig. 4 by employing the notion of *strictness variables*, ranging over 0,1. A *pre-strictness type* is now a strictness type where the 0/1's have been replaced by strictness variables. Then judgements take the form $\Gamma \vdash e : t, W, C$ with t a pre-strictness type, with Γ mapping (program) variables into pre-strictness types, with e an expression, with W a mapping from (program) variables into strictness variables, and with C a set of *constraints* among the strictness variables.

The crucial point is that given a proof tree one can *normalize* the constraints by traversing the tree from leaves to root. Employing the convention that $\mathbf{b}^+$ denotes the strictness variables occurring in covariant position in the actual judgement, that $\mathbf{b}^-$ denotes the strictness variables occurring in contravariant position in the actual judgement and that $\mathbf{b}_0$ denotes the strictness variables *not* occurring in the actual judgement but "higher up in the proof tree", a normalized set of constraints consists of

– A constraint of form $\mathbf{b}^+ \geq g(\mathbf{b}^-)$, with g a monotone function from $\{0,1\}^n$ into $\{0,1\}^m$ for appropriate n,m. Hence we see that for each choice of assignment to $\mathbf{b}^-$ there exists a least assignment to $\mathbf{b}^+$.
– A constraint of form $\mathbf{b}_0 \gg g_0(\mathbf{b}^-)$, with g_0 a monotone function. The interpretation of the strange symbol $\gg$ is that by replacing it by "=" one surely gets a solution to the inference system in Fig. 4; and all solutions to this inference system satisfy the constraint resulting from replacing $\gg$ by $\geq$.

4 An Algorithm for Thunkification

4.1 The Mapping $C_t^{t'}$

The first step will be, for types t and t' such that $t \leq t'$, to define a mapping $C_t^{t'}$ from expressions into expressions. The translation is motivated by the desire that if e has type $Z(t)$, then $C_t^{t'}(e)$ has type $Z(t')$.

Example 3. Suppose $t = \mathsf{Int}{\rightarrow_0}\mathsf{Int}$ and $t' = \mathsf{Int}{\rightarrow_1}\mathsf{Int}$, and suppose e has type $\mathsf{Int}{\rightarrow}\mathsf{Int}$. $C_t^{t'}$ then has to translate e into something of type $[\mathsf{Int}]{\rightarrow}\mathsf{Int}$ – it is easily seen that $\lambda x.e\ \mathcal{D}(x)$ (with x fresh) will do the job.

$C_t^{t'}$ is defined as follows (inductively in the "size" of t and t'):

1. If $t = t'$, then $C_t^{t'}(e) = e$.
2. If $t = t_1{\rightarrow_0}t_2$ and $t' = t'_1{\rightarrow_0}t'_2$ (with $t'_1 \leq t_1$, $t_2 \leq t'_2$), then (where x is a "fresh" variable)
$$C_t^{t'}(e) = \lambda x.C_{t_2}^{t'_2}(e\ C_{t'_1}^{t_1}(x))$$
3. If $t = t_1{\rightarrow_1}t_2$ and $t' = t'_1{\rightarrow_1}t'_2$ (with $t'_1 \leq t_1$, $t_2 \leq t'_2$), then (where x is a "fresh" variable)
$$C_t^{t'}(e) = \lambda x.C_{t_2}^{t'_2}(e\ \underline{C_{t'_1}^{t_1}(\mathcal{D}(x))})$$
4. If $t = t_1{\rightarrow_0}t_2$ and $t' = t'_1{\rightarrow_1}t'_2$ (with $t'_1 \leq t_1$, $t_2 \leq t'_2$), then (where x is a "fresh" variable)
$$C_t^{t'}(e) = \lambda x.C_{t_2}^{t'_2}(e\ C_{t'_1}^{t_1}(\mathcal{D}(x)))$$

4.2 The Translation from CBN to CBV

Given an expression e, and a proof of $\Gamma, T{\vdash_{\mathrm{sa}}}e : t, W$. We now present an algorithm for transforming e into an expression e', with the aim that the "CBV-semantics" of e' should equal the "CBN-semantics" of e.

The translation is defined inductively in the proof tree – several cases:

– Suppose $\Gamma, T{\vdash_{\mathrm{sa}}}e : t', W'$ because $\Gamma, T{\vdash_{\mathrm{sa}}}e : t, W$ and $t \leq t'$, $W' \subseteq W$. Suppose e (by the latter proof tree) transforms into e'. Then e (by the former proof tree) transforms into $C_t^{t'}(e')$.
– Suppose $e = c$, and $\Gamma, T{\vdash_{\mathrm{sa}}}e : CT_{\mathrm{sa}}(c), \emptyset$. Then we let $e' = c$.
– Suppose $e = x$, and $\Gamma, T{\vdash_{\mathrm{sa}}}e : \Gamma(x), \{x\}$. Two cases:
  - If $x \in T$, we let $e' = \mathcal{D}(x)$ (as x will be bound to a thunkified argument).
  - If $x \notin T$, we let $e' = x$.
– Suppose $e = \lambda x.e_1$, and suppose e_1 (using the relevant proof tree) translates into e'_1. Then e translates into $\lambda x.e'_1$.
– Suppose $e = e_1 e_2$, and suppose e_1 and e_2 (using the relevant proof trees) translate into e'_1 resp. e'_2. Two cases:
  - If e_1 is of type $t_2{\rightarrow_0}t_1$, e translates into $e'_1 e'_2$.
  - If e_1 is of type $t_2{\rightarrow_1}t_1$, e translates into $e'_1\underline{e'_2}$.

– Suppose $e = \mathsf{if}\ e_1\ e_2\ e_3$, and suppose e_1, e_2 and e_3 (using the relevant proof trees) translate into e_1', e_2' resp. e_3'. Then e translates into $\mathsf{if}\ e_1'\ e_2'\ e_3'$.
– Suppose $e = \mathsf{rec}\ f\ e_1$ (resp. $\mathsf{rec}_n\ f\ e_1$), and suppose e_1 (using the relevant proof tree) translates into e_1'. Then e translates into $\mathsf{rec}\ f\ e_1'$ (resp. $\mathsf{rec}_n\ f\ e_1'$).

This is similar to the translation produced by the thunkification algorithm from [DH93].

Example 4. Consider the expression $twice = \lambda f.\lambda x.f(fx)$. $twice$ has strictness type $(\mathsf{Int}\rightarrow_1\mathsf{Int})\rightarrow_0(\mathsf{Int}\rightarrow_1\mathsf{Int})$ because

$$\{f := \mathsf{Int}\rightarrow_1\mathsf{Int}\}, \emptyset \vdash_{\mathrm{sa}} \lambda x.f(fx) : \mathsf{Int}\rightarrow_1\mathsf{Int}, \{f\} \text{ and}$$

$$\{f := \mathsf{Int}\rightarrow_1\mathsf{Int}, x := \mathsf{Int}\}, \{x\} \vdash_{\mathrm{sa}} f(fx) : \mathsf{Int}, \{f\} \text{ etc.}$$

Accordingly, $twice$ translates into the term

$$\lambda f.\lambda x.f\underline{f\underline{\mathcal{D}(x)}}$$

of type $([\mathsf{Int}]\rightarrow\mathsf{Int})\rightarrow([\mathsf{Int}]\rightarrow\mathsf{Int})$. We see that there is room for some (peephole) optimization here, as $\underline{\mathcal{D}(x)}$ could be replaced by x.

Notice that $twice$ also has strictness type $(\mathsf{Int}\rightarrow_0\mathsf{Int})\rightarrow_0(\mathsf{Int}\rightarrow_0\mathsf{Int})$. Using the corresponding proof tree, $twice$ just translates into itself.

5 Correctness Predicates

We now embark on expressing the *correctness* of the translation – something not addressed in [DH93]. As a first step, we consider closed expressions only – to this end we define a predicate $\sim_t$, indexed over strictness types, such that $q\sim_t q'$ is defined whenever q is a closed expression of type $\mathsf{E}(t)$, and q' is a closed expression of type $Z(t)$. $\sim_t$ is defined inductively on t:

– $q\sim_{\mathsf{Base}}q'$ holds iff for all constants c we have $q\Rightarrow_N^* c$ iff $q'\Rightarrow_V^* c$ (in particular, q loops by CBN iff q' loops by CBV).
– $q_1\sim_{t_1\rightarrow_0 t_2}q_1'$ holds iff for all q_2,q_2' such that $q_2\sim_{t_1}q_2'$ we have $q_1q_2\sim_{t_2}q_1'q_2'$.
– $q_1\sim_{t_1\rightarrow_1 t_2}q_1'$ holds iff for all q_2,q_2' such that $q_2\sim_{t_1}q_2'$ we have $q_1q_2\sim_{t_2}q_1'\underline{q_2'}$.

This very much resembles a logical relation, but notice the difference between $\sim_{t_1\rightarrow_0 t_2}$ and $\sim_{t_1\rightarrow_1 t_2}$. Thus the predicate closely reflects how expressions are to be translated, cf. the claim in [Wan93]:

> This work suggests that the proposition associated with a flow analysis can simply be that "the optimization works".

Now we are ready to consider arbitrary (non-closed) expressions. The main correctness predicate takes the form $e\ \mathsf{COR}(t, W, \Gamma, T)\ e'$, where e and e' are expressions, t belongs to $\mathcal{T}_{\mathrm{sa}}$, Γ maps variables into $\mathcal{T}_{\mathrm{sa}}$, and W and T are sets of variables. We shall need an auxiliary function Z_T, mapping from $\mathcal{T}_{\mathrm{sa}}$-environments into $\mathcal{T}$-environments: $Z_T(\Gamma)(x) = Z(\Gamma(x))$ for $x \notin T$;
and $Z_T(\Gamma)(x) = [Z(\Gamma(x))]$ for $x \in T$.

Definition 5. e $\mathsf{COR}(t, W, \Gamma, T)$ e' holds iff (with $\{x_1 \dots x_n\}$ being the domain of Γ)

1. $\Gamma, T \vdash_{\mathrm{sa}} e : t, W$.
2. $Z_T(\Gamma) \vdash e' : Z(t)$.
3. $\mathsf{FV}(e) = \mathsf{FV}(e')$.
4. Let closed terms q_i, q'_i $(i \in \{1 \dots n\})$ be such that $q_i \sim_{\Gamma(x_i)} q'_i$. Then
$$e[\{q_1 \dots q_n\}/\{x_1 \dots x_n\}] \sim_t e'[\{Q'_1 \dots Q'_n\}/\{x_1 \dots x_n\}]$$
where Q'_i is defined as follows: if $x_i \in T$ then $Q'_i = \underline{q'_i}$ else $Q'_i = q'_i$.
Moreover, suppose that i is such that $x_i \in W$ and $q_i \sim_{\Gamma(i)} \Omega$. Then
$$e[\{q_1 \dots q_n\}/\{x_1 \dots x_n\}] \sim_t \Omega$$

The first part of 4 resembles the standard way of extending relations from closed terms to open terms; the second part of 4 expresses that the variables in W are "needed".

5.1 Correctness Theorems

We have the following theorem, to be proved in [Amt93]:

Theorem 6. *Suppose $\Gamma, T \vdash_{sa} e : t, W$, suppose e contains no unbounded recursion (i.e. only rec_n's and no rec's) and suppose e (by means of the corresponding proof tree) translates into e'. Then e $\mathsf{COR}(t, W, \Gamma, T)$ e'.*

The restriction to bounded recursion is motivated by the SOS-rule
$\mathsf{rec}\ f\ e \Rightarrow_N e[(\mathsf{rec}\ f\ e)/f]$, as we want to (inductively) use properties of the latter rec to prove properties of the former rec.

The proof of Theorem 6 proceeds roughly speaking as follows:

1. A number of properties of $\sim_t$ are proved (by induction on t). For instance, we have that if $q \Rightarrow_N q_1$ and $q \sim_t q'$ then also $q_1 \sim_t q'$.
2. Some properties of $C_t^{t'}$ are formulated and proved – for instance that if $q \sim_t q'$ then $q \sim_{t'} C_t^{t'}(q')$.
3. Finally, we are able to prove Theorem 6 by induction in the proof tree.

By means of Theorem 6 we can prove what we are really looking for:

Theorem 7. *Suppose q is a closed expression (which may contain unbounded recursion) such that $\emptyset, \emptyset \vdash_{sa} q : \mathsf{Base}, \emptyset$. Let q' be the translation of q, using the algorithm in Sect. 4. Now for all constants (of base type) c, $q \Rightarrow_N^* c$ iff $q' \Rightarrow_V^* c$.*

Proof. First some notation: given n, let q_n be the result of substituting rec_n for all occurrences of rec. It is easy to see that q_n translates into q'_n.

First (the "only if" part) suppose $q \Rightarrow_N^* c$. It is easy to see that there exists n such that $q_n \Rightarrow_N^* c$. Since q_n and q'_n does not contain unbounded recursion, Theorem 6 tells us that q_n $\mathsf{COR}(\mathsf{Base}, \emptyset, \emptyset, \emptyset)$ q'_n. This implies that $q_n \sim_{\mathsf{Base}} q'_n$, so $q'_n \Rightarrow_V^* c$. But then it is immediate that $q' \Rightarrow_V^* c$.

The "if" part is analogous. □

6 Concluding Remarks

We have presented a type system for strictness analysis, presented an algorithm which translates a CBN-term into an equivalent CBV-term and finally given a proof of the correctness of the analysis/translation.

It may be of interest to investigate closer the power of our strictness analysis, relative to other approaches. And in order to avoid the kind of superfluous dethunkification/thunkification we encountered in Example 4, one may consider keeping track of context – somewhat similar to what is done in [NN90].

References

[Amt93] Torben Amtoft. Strictness types: An inference algorithm and an application. Technical Report PB-448, DAIMI, University of Aarhus, Denmark, August 1993.

[BHA86] Geoffrey L. Burn, Chris Hankin, and Samson Abramsky. Strictness analysis for higher-order functions. *Science of Computer Programming*, 7:249–278, 1986.

[DH92] Olivier Danvy and John Hatcliff. Thunks (continued). In M. Billaud et al., editor, *Analyse statique, Bordeaux 92 (WSA '92)*, pages 3–11, September 1992.

[DH93] Olivier Danvy and John Hatcliff. CPS transformation after strictness analysis. *ACM Letters on Programming Languages and Systems*, 1(3), 1993.

[Hug89] John Hughes. Why functional programming matters. *Computer Journal*, 32(2):98–107, 1989.

[Jen91] Thomas P. Jensen. Strictness analysis in logical form. In John Hughes, editor, *International Conference on Functional Programming Languages and Computer Architecture*, pages 352–366. Springer Verlag, LNCS 523, August 1991.

[KM89] Tsung-Min Kuo and Prateek Mishra. Strictness analysis: A new perspective based on type inference. In *International Conference on Functional Programming Languages and Computer Architecture*, pages 260–272. ACM Press, September 1989.

[Lan92] Torben Poort Lange. The correctness of an optimized code generation. Technical Report PB-427, DAIMI, University of Aarhus, Denmark, November 1992. Also in the proceedings of PEPM '93, Copenhagen, ACM press.

[Myc80] Alan Mycroft. The theory of transforming call-by-need to call-by-value. In B. Robinet, editor, *International Symposium on Programming, Paris*, pages 269–281. Springer Verlag, LNCS 83, April 1980.

[NN90] Hanne Riis Nielson and Flemming Nielson. Context information for lazy code generation. In *ACM Conference on Lisp and Functional Programming*, pages 251–263. ACM Press, June 1990.

[Wan93] Mitchell Wand. Specifying the correctness of binding-time analysis. In *ACM Symposium on Principles of Programming Languages*, pages 137–143. ACM Press, January 1993.

[Wri91] David A. Wright. A new technique for strictness analysis. In *TAPSOFT 91*, pages 235–258. Springer Verlag, LNCS 494, April 1991.

[Wri92] David A. Wright. An intensional type discipline. *Australian Computer Science Communications*, 14, January 1992.