Strictness Types: An Inference Algorithm and an Application

Torben Amtoft
Computer Science Department
Aarhus University, Ny Munkegade, DK-8000 Arhus C,
Denmark
internet: tamtoft@daimi.aau.dk

August 10, 1993

Abstract

This report deals with *strictness types*, a way of recording whether a function needs its argument(s) or not. We shall present an inference system for assigning strictness types to expressions and subsequently we transform this system into an *algorithm* capable of annotating expressions with strictness types. We give an example of a transformation which can be optimized by means of these annotations, and finally we prove the correctness of the optimized transformation — at the same time proving the correctness of the annotation.

Everything has been implemented; documentation can be found in appendix.

1 Introduction

1.1 Strictness Types

Strictness analysis is the task of detecting whether a function needs its arguments. Recent years have seen many approaches to strictness analysis,

most based on abstract interpretation – the starting point being the work of Mycroft [Myc80] which was extended to higher order functions by Burn, Hankin and Abramsky [BHA86]. These analyses have been proved correct in the following sense: if $f^{\#}$ is the abstract denotation of the function f, and if $f^{\#}(0) = 0$ (0 denoting the bottom value in the abstract domain), then $f(\bot) = \bot$ (in the concrete domain) – that is, the call of f will not terminate if its argument does not terminate[1].

Kuo and Mishra [KM89] presented a type system where types t are formed from 0 (denoting non-termination), 1 (denoting non-termination or termination, i.e. any term) and $t_1 \to t_2$. Accordingly, if it is possible to assign a function the type $0 \to 0$ we know that the function is strict. This system, however, is strictly weaker than one based on abstract interpretation – on the other hand, Jensen [Jen91] proves that if conjunction types are added to the type system one gets a type system with the same power as abstract interpretation. To get an intuitive feeling of what's going on, consider a function f with abstract denotation defined by $f(0)(1) = 0,\ f(1)(0) = 0,\ f(1)(1) = 1$ (and by monotonicity hence also $f(0)(0) = 0$). Such a function needs *both* of its arguments, and accordingly it has type $0 \to 1 \to 0$ *as well as* type $1 \to 0 \to 0$.

Wright has proposed alternative type systems [Wri91] and [Wri92]. The idea is to annotate the *arrows*: if a function can be assigned type $\mathsf{Int} \to_0 \mathsf{Int}$ this means that the function is strict, whereas a type $\mathsf{Int} \to_1 \mathsf{Int}$ doesn't tell anything about strictness properties. It should be clear that this is weaker than abstract interpretation, as $\mathsf{Int} \to_0 \mathsf{Int}$ corresponds to two functions on the abstract domain: the identity function and the zero function. On the other hand, in some cases the method is more powerful than the one of [KM89]: a function which needs both of its arguments (cf. the above) can be assigned type $\mathsf{Int} \to_0 \mathsf{Int} \to_0 \mathsf{Int}$.

In Sect. 3 we are going to present a type inference system based on Wright's idea, and by means of a few examples we shall illustrate the strengths and weaknesses of the system. A proof of the "semantic correctness" of the inference system will not appear before Sect. 6 where we shall prove the *overall* correctness of the system and a transformation exploiting the strictness information.

[1]This is not exactly equivalent to saying that "f needs its argument", as f may loop without ever looking at its argument.

An ordering on strictness types, monotone in covariant position and anti-monotone in contravariant position, will be imposed. Thus e.g. $(\mathsf{Int} \rightarrow_1 \mathsf{Int}) \rightarrow_0 \mathsf{Int} \leq (\mathsf{Int} \rightarrow_0 \mathsf{Int}) \rightarrow_1 \mathsf{Int}$. So if $t_1 \leq t_2$ then t_1 carries more information than t_2. The inference system will have a "subsumption rule", stating that if an expression has a type then it also has any greater type – thus allowing one to "forget information".

1.2 An Inference Algorithm

Section 4 is devoted to transforming the type inference system from Sect. 3 into an algorithm. As we want to concentrate upon strictness aspects and not upon type inference in general, we shall assume that the underlying type (such as e.g. $\mathsf{Int} \rightarrow \mathsf{Int}$) has been given in advance. The first step is to "inline" the subsumption rule into the other rules, thus making the system "syntax-directed". Next we rewrite the system into one using constraints. As a result we get a system with the property that for any expression e it is straight-forward to assign e a typing where the arrows are annotated by *strictness variables* (variables which can assume the values 0 and 1), at the same time generating a set of constraints among these strictness variables. We are thus left with the problem of solving those constraints.

In type inference one is usually interested in finding a "principal type" such that all other valid typings can be found as substitution instances of this type. This is also the approach taken in [Wri91]. In our framework (which in this respect differs from Wright's), we would like to find a "least type" (wrt. the ordering $\leq$). However, in general no least typing exists as can be seen from the term $\mathsf{twice} = \lambda f.\lambda x.f(f(x))$ which 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, this example suggests that for each choice of assignments to the arrows occurring in *contravariant* positions there exists a least assignment to the arrows occurring in *covariant* position. This motivates the definition of a *normalized* set of constraints, which (loosely speaking) is a constraint set where each constraint is of form $\vec{b}^+ \geq g(\vec{b}^-)$ where $\vec{b}^+(\vec{b}^-)$ are strictness variables occurring in covariant (contravariant) position, and where g is a monotone function.

We shall see that it is possible to normalize a constraint set "on the

fly", i.e. during a traversal from leaves to root in the proof tree. Thus the conjecture above is true; once the annotations of arrows in contravariant position is fixed there exists a least annotation of the covariant arrows.

The usual approach to constraint solving is to collect them all and then solve them – which approach is the best one *algorithmically* is hard to say; we shall not address this issue.

The normalization algorithm employs some techniques which we think might be new – on the other hand, since constraint solving appears in numerous contexts it is quite possible that similar approaches exist in the literature.

1.3 Tanslating from CBN to CBV

Call-by-name (CBN) evaluation of the λ-calculus (and especially the variant known as "lazy evaluation") has been widely praised (e.g. in [Hug89]) because it makes programming a much more convenient task (another advantage is that referential transparency holds). On the other hand, as most implementations are call-by-value (CBV) one has to find means for translating from CBN to CBV. The naive approach (as presented e.g. in [DH92]) is to "thunkify" all arguments to applications, 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".

This is clearly suboptimal since if e_1 is strict there is no need to thunkify its argument – this observation being the motivation for e.g. Mycroft's work on strictness analysis. Accordingly, in Sect. 5 we shall present a translation from CBN to CBV which exploits the information present in the strictness types. This translation is essentially similar to the one given by Danvy and Hatcliff [DH93].

1.4 Proving the Tanslation Correct

The optimized translation from CBN to CBV is folklore – but a correctness proof is certainly not. For instance, the translation presented in [DH93] is not proved correct, and even though the strictness analysis presented in [BHA86] is proved correct (in the sense that the abstract semantics actually abstracts the concrete semantics) the correctness of an optimization based on this analysis has not been proved. The same remarks apply to e.g. [Wri91] and reflect a quite general phenomenon, cf. the claims made in [Wan93, p. 137]:

> 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.

In [Wan93], Wand proved the correctness of a partial evaluator which exploits binding time information. In Sect. 6 we follow this trend, within the context defined in Sect. 5 (i.e. a CBN-to-CBV translator exploiting strictness information). Also something similar can be found in [Lan92] where the correctness of a code generation exploiting strictness information is proved.

The basic idea in expressing the correctness of the translation from Sect. 5 is to use logical relations (on closed terms): $q \sim_t q'$ should be interpreted as stating that q' is a correct translation of q. If t is a base type, $q \sim_t q'$ holds iff q when evaluated by CBN yields the same result as q' when evaluated by CBV (in particular q loops by CBN iff q' loops by CBV). For a strict function type $t = t_1 \to_0 t_2, q \sim_t q'$ means that whenever $q_1 \sim_{t_1} q_1'$; then $qq_1 \sim_{t_2} q'q_1'$. For a non-strict function type $t = t_1 \to t_2, q \sim_{t_1} q'$ means that whenever $q_1 \sim_{t_1} q_1'$ then $qq_1 \sim_{t_2} q'(\lambda x.q_1')$ (x a fresh variable).

The noteworthy point is that the fact that a function actually *is* strict is not expressed using some relationship between concrete/abstract domains, but simply by stating that it is correct *not* to thunkify its argument! This corresponds to the claim in [Wan93, p. 137]:

> This work suggests that the proposition associated with a flow

analysis can simply be that "the optimization works".

The extension of the correctness predicate to open terms is rather straightforward – and so is the correctness proof, the only tricky point being how to cope with recursion.

1.5 An Implementation

The type inference algorithm from Sect. 4 and the translation algorithm from Sect. 5 has been implemented in Miranda[2]. The user interface is as follows:

- the user writes a λ-expression, and provides the underlying type of the bound variables;
- the user provides the annotation of the contravariant arrows in the overall type;
- then the system produces the least valid annotation of the remaining arrows, and translates the original expression into a CBV-equivalent expression.

The system is documented in Appendix A.

1.6 Acknowledgements

The author is 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.

[2]Miranda is a trademark of Research Software Limited.

2 Preliminaries

Expressions

An expression is either a constant c; a variable x; an abstraction $\lambda x.e$; an application $e_1 e_2$; a conditional $\mathsf{if}\ e_1\ e_2\ e_3$ or a recursive definition $\mathsf{rec}\ f\ e$.

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 (underlying) 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 (we will expect to have $Ct(7) = \mathsf{Int}$, $Ct(+) = \mathsf{Int} \to \mathsf{int} \to \mathsf{Int}$ etc.).

$$\Gamma \vdash c : Ct(c) \qquad (\Gamma_1, (x : t), \Gamma_2) \vdash x : t$$

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

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

Figure 1: An inference system for (underlying) types.

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; Γ will be represented as a *list* of pairs of form $(x \ : \ t)$[3][4]. 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'.

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 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 $Ct(c_1) = \mathsf{Base} \to t$, $Ct(c_2) = \mathsf{Base}$ 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. $\mathsf{Applycon}(/_7, 0) = (/_7, 0)$.

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 need the extra assumption that if $Ct(c) = \mathsf{Bool}$ then $c = \mathsf{True}$ or $c = \mathsf{False}$.

Fact 2.1 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 $\Omega = \mathsf{ret}\ f\ f$. 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 $\Gamma \vdash \Omega \ : \ t$.

[3]To concatenate e.g. the list l_1 with the list l_2 we shall write (l_1, l_2).

[4]In case of multiple occurrences of some variable x, it is the leftmost occurrence which "counts" – we are not going to bother further about that.

$$(\lambda x.e)q \Rightarrow_N e[q/x] \qquad \frac{q_1 \Rightarrow_N q_1'}{q_1 q_2 \Rightarrow_N q_1' q_2}$$

$$c_1 c_2 \Rightarrow_N \mathsf{Applycon}(c_1, c_2) \qquad \frac{q_2 \Rightarrow_N q_2'}{c q_2 \Rightarrow_N c q_2'}$$

$$\mathsf{if\ True}\ q_2\ q_3 \Rightarrow_N q_2 \qquad \mathsf{if\ False}\ q_2\ q_3 \Rightarrow_N q_3$$

$$\frac{q_1 \Rightarrow_N q_1'}{\mathsf{if}\ q_1\ q_2\ q_3 \Rightarrow_N \mathsf{if}\ q_1'\ q_2\ q_3} \qquad \mathsf{rec}\ f\ e \Rightarrow_N e[(\mathsf{rec}\ f\ e)/f]$$

Figure 2: A SOS for CBN.

$$(\lambda x.e)q \Rightarrow_V e[q/x],\ \text{if } q \text{ in } \mathsf{WHNF} \qquad \frac{q_1 \Rightarrow_V q_1'}{q_1 q_2 \Rightarrow_V q_1' q_2}$$

$$c_1 c_2 \Rightarrow_V \mathsf{Applycon}(c_1, c_2) \qquad \frac{q_2 \Rightarrow_V q_2'}{q_1 q_2 \Rightarrow_V q_1 q_2'},\ \text{if } q_1 \text{ in } \mathsf{WHNF}$$

$$\mathsf{if\ True}\ q_2\ q_3 \Rightarrow_V q_2 \qquad \mathsf{if\ False}\ q_2\ q_3 \Rightarrow_V q_3$$

$$\frac{q_1 \Rightarrow_V q_1'}{\mathsf{if}\ q_1\ q_2\ q_3 \Rightarrow_V \mathsf{if}\ q_1'\ q_2\ q_3} \qquad \mathsf{rec}\ f\ e \Rightarrow_V e[(\mathsf{rec}\ f\ e)/f]$$

Figure 3: A SOS for CBV.

Thunkification and Dethunkification

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

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 2.2 If $\Gamma \vdash e\ :\ t$, then $\Gamma \vdash c\ :\ [t]$. If $\Gamma \vdash \underline{e}\ :\ [t]$, 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

In this section we shall augment types with strictness information, that is annotate the arrows. The set of strictness types, $\mathcal{T}_{sa}$, 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).

It will be convenient to introduce some notation: if t is a (standard) type in $\mathcal{T}$, and if $\vec{b}^+$ and $\vec{b}^-$ are vectors of 0 or 1's, then $t[\vec{b}^+, \vec{b}^-]$ denotes t where all covariant arrows are marked (from left to right) as indicated by $\vec{b}^+$ and where all contravariant arrows are marked (from left to right) as indicated by $\vec{b}^-$. More formally, we have

- $\mathsf{Base}[(),\ ()] = \mathsf{Base}$;
- $(t_1 \rightarrow t_2)[(\vec{b}_1^+, b^+, \vec{b}_2^+), (\vec{b}_1^-, \vec{b}_2^-)] = t_1[\vec{b}_1^-, \vec{b}_1^+] \rightarrow_{b^+ t_2[\vec{b}_2^+, \vec{b}_2^-]}$

Example 3.1 With $t = ((\mathsf{Int} \rightarrow \mathsf{Int}) \rightarrow \mathsf{Int}) \rightarrow ((\mathsf{Int} \rightarrow \mathsf{Int}) \rightarrow \mathsf{Int})$, we have

$$t[(b_1, b_2, b_3), (b_4, b_5)] = ((\mathsf{Int} \rightarrow_{b_1} \mathsf{Int}) \rightarrow_{b_4} \mathsf{Int}) \rightarrow_{b_2} ((\mathsf{Int} \rightarrow_{b_5} \mathsf{Int}) \rightarrow_{b_3} \mathsf{Int})$$

□

If $\Gamma = ((x_1\ :\ t_1) \ldots (x_n\ :\ t_n))$ then

$$\Gamma[(\vec{b}_1^-, \ldots, \vec{b}_n^-), (\vec{b}_1^+, \ldots, \vec{b}_n^+)] = ((x_1\ :\ t_1[\vec{b}_1^-, \vec{b}_1^+]), \ldots, (x_n\ :\ t_n[\vec{b}_n^-, \vec{b}_n^+]))$$

An Ordering Relation

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 $\mathsf{Int} \to_0 \mathsf{Int}$ than to know that it is of type $\mathsf{Int} \to_1 \mathsf{Int}$. Similarly, it is more informative to know that a function is of type $(\mathsf{Int} \to_1 \mathsf{Int}) \to_0 (\mathsf{Int} \to_0 \mathsf{Int})$ (it maps *arbitrary* functions into strict functions) than to know that is is of type $(\mathsf{Int} \to_0 \mathsf{Int}) \to_0 (\mathsf{Int} \to_0 \mathsf{Int})$ (it maps strict functions into strict functions).

Fact 3.2 $t[\vec{b}_1^+, \vec{b}_1^-] \leq t[\vec{b}_2^+, \vec{b}_2^-]$ iff $\vec{b}_1^+ \leq \vec{b}_2^+$ and $\vec{b}_2^- \leq \vec{b}_1^-$ (pointwise).

The Relation Between $\mathcal{T}$ and $\mathcal{T}_{sa}$

We define a mapping E from $\mathcal{T}_{sa}$ into $\mathcal{T}$, which just removes annotations from arrows – that is, we have $\mathsf{E}(\mathsf{Base}) = \mathsf{Base}$, $\mathsf{E}(t_1 \to_0 t_2) = \mathsf{E}(t_1) \to \mathsf{E}(t_2)$, $\mathsf{E}(t_1 \to_1 t_2) = \mathsf{E}(t_1) \to \mathsf{E}(t_2)$. Clearly, $\mathsf{E}(t[\vec{b}^+, \vec{b}^-]) = t$. And if $t_1 \leq t_2$, then $\mathsf{E}(t_1) = \mathsf{E}(t_2)$.

We have to extend Ct into CT_{sa} a mapping from constants into strictness types, and doing so we shall exploit that all constants are strict (recall that the non-strict constant if has been given a special status). Accordingly, we define $CT_{sa}(c) = Ct(c)[\vec{0}, ()]$.

The Inference System

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

- Γ is an environment (represented as a list) assigning strictness types to variables;
- e is an expression such that if x is a free variable of e ($x \in \mathsf{FV}(e)$) then $\Gamma(x)$ is defined;
- t is a strictness type;
- W maps the domain of Γ into $\{0, 1\}$. It may be helpful to think of W as follows: if $W(x) = 0$ then x is needed in order to evaluate e to "head normal form".

Some notation: if $\Gamma = ((x_1 \ : \ t_1) \ldots (x_n \ : \ t_n))$ and if $W(x_i) = b_i$ we shall often write $W = (b_1 \ldots b_n)$. Also, we shall sometimes write $W = \{x_i | b_i = 0\}$ (i.e. identify W with the set of variables on which W assumes the value 0). In the natural way, E is extended to work on environments.

Now a brief explanation of the inference system: the first inference rule (the "subsumption 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 rule for variables expresses (among other things) that in order to evaluate x it is necessary to evaluate x (!) but no other variables are needed. The rule for abstractions (among other things) says 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 rule for applications (among other things) says that the variables needed to evaluate e_1 are also needed to evaluate $e_1 e_2$; and if e_1 is strict then the variables needed to evaluate e_2 will also be needed to evaluate $e_1 e_2$. The rule for conditionals (among other things) says that if a variable is needed to evaluate the test then it is also needed to evaluate the whole expression; and also if a variable is needed in order to evaluate *both* branches it will be needed to evaluate the whole expression.

Notice that $() \vdash_{sa} \Omega \ : \ t, \ ()$ for all strictness types t.

An expression which can be assigned a strictness type can also be assigned an underlying type:

Fact 3.3 Suppose $\Gamma \vdash_{sa} e \ : \ t, W$. Then $\mathsf{E}(\Gamma) \vdash e \ : \mathsf{E}(t)$.

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

Fact 3.4 Suppose $\Gamma \vdash e \ : \ t$. Then $\Gamma[\vec{1}, \vec{1}] \vdash_{sa} e \ : \ t[\vec{1}, \vec{1}], \vec{1}$.

PROOF: An easy induction in the proof tree. In the case of a constant c, we have to use the subsumption rule and exploit that $CT_{sa}(c) = Ct(c)[\vec{0}, ()] \leq Ct(c)[\vec{1}, ()]$. $\Box$

$$\frac{\Gamma \vdash_{sa} e : t, W}{\Gamma \vdash_{sa} e : t', W'} \text{ if } t' \geq t, W' \geq W$$

$$\Gamma \vdash_{sa} c : CT_{sa}(c), \vec{1}$$

$$(\Gamma_1, (x : t), \Gamma_2) \vdash_{sa} x : t, (\vec{1}, 0, \vec{1})$$

$$\frac{((x : t_1), \Gamma) \vdash_{sa} e : t, (b, W)}{\Gamma \vdash_{sa} \lambda x.e : t_1 \rightarrow_b t, W}$$

$$\frac{\Gamma \vdash_{sa} e_1 : t_2 \rightarrow_b t_1, W_1 \ \Gamma \vdash_{sa} e_2 : t_2, W_2}{\Gamma \vdash_{sa} e_1 e_2 : t_1, W}$$
$$\text{if } W(x) = W_1(x) \sqcap (b \sqcup W_2(x)) \text{ for all } x$$

$$\frac{\Gamma \vdash_{sa} e_1 : \mathsf{Bool}, W_1 \ \Gamma \vdash_{sa} e_2 : t, W_2 \ \Gamma \vdash_{sa} e_3 : t, W_3}{\Gamma \vdash_{sa} (\mathsf{if}\ e_1\ e_2\ e_3) : t, W}$$
$$\text{if } W(x) = W_1(x) \sqcap (W_2(x) \sqcup W_3(x)) \text{ for all } x$$

$$\frac{((f : t), \Gamma) \vdash_{sa} e : t, (b, W)}{\Gamma \vdash_{sa} (\mathsf{rec}\ f\ e) : t, W}$$

Figure 4: An inference system for strictness types.

Example 3.5 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

$$() \vdash_{sa} \mathsf{rec}\ f\ \lambda x.\lambda y.\lambda z.e : \mathsf{Int} \to_0 \mathsf{Int} \to_0 \mathsf{Int} \to_0 \mathsf{Int}, \emptyset$$

This is because we – with $\Gamma_1 = ((f : \mathsf{Int} \to_0 \mathsf{Int} \to_0 \mathsf{Int} \to_0 \mathsf{Int}))$ – have

$$\Gamma_1 \vdash_{sa} \lambda x.\lambda y.\lambda z.e : \mathsf{Int} \to_0 \mathsf{Int} \to_0 \mathsf{Int} \to_0 \mathsf{Int}, \emptyset$$

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

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

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

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

The latter follows since e.g. $\Gamma_2 \vdash_{sa} (f\ y) : \mathsf{Int} \to_0 \mathsf{Int} \to_0 \mathsf{Int}, \{y\}$. □

Example 3.6 Our analysis is not very good at handling recursive definitions with free variables. To see this, consider the function g given

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

Clearly ge_1e_2 will loop if e_1 loops, so the analysis *ought* to conclude that g has strictness type $\mathsf{Int} \to_0 \mathsf{Int} \to_0 \mathsf{Int}$. However, we can do no better than inferring that g has strictness type $\mathsf{Int} \to_1 \mathsf{Int} \to_0 \mathsf{Int}$ – this is because it is impossible to deduce $\ldots \vdash_{sa} (\mathsf{rec}\ f\ \ldots) : \ldots, \{y\}$ which in turn is because it is impossible to deduce $\ldots \vdash_{sa} (\mathsf{if}\ (x = 0)\ y\ (f\ (x - 1))) : \ldots, \{x, y\}$. 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. □

4 An Inference Algorithm

In this section we shall work on the inference system from Fig. 4 and transform it into an algorithm. This will be a two-stage process: first the inference system is transformed into an equivalent one using constraints; then an algorithm is given for solving those constraints. We shall assume that all subexpressions are (implicitly) annotated with an "underlying type" (belonging to $\mathcal{T}$) such that the expression is well-typed according to the rules in Fig. 1.

$$\Gamma \vdash_{sa} c : t, \vec{1} \text{ if } t \geq CT_{sa}(c)$$

$$(\Gamma_1, (x : t), \Gamma_2) \vdash_{sa} x : t', (\vec{1}, b, \vec{1}) \text{ if } t' \geq t, b \geq 0$$

$$\frac{((x : t_1), \Gamma) \vdash_{sa} e : t, (b, W)}{\Gamma \vdash_{sa} \lambda x.e : t_1 \rightarrow_b t, W}$$

$$\frac{\Gamma \vdash_{sa} e_1 : t_2 \rightarrow_b t_1, W_1 \ \Gamma \vdash_{sa} e_2 : t_2, W_2}{\Gamma \vdash_{sa} e_1 e_2 : t_1, W} \\ \text{if } W(x) \geq W_1(x) \sqcap (b \sqcup W_2(x)) \text{ for all } x$$

$$\frac{\Gamma \vdash_{sa} e_1 : \mathsf{Bool}, W_1 \ \Gamma \vdash_{sa} e_2 : t_2, W_2 \ \Gamma \vdash_{sa} e_3 : t_3, W_3}{\Gamma \vdash_{sa} (\mathsf{if}\ e_1\ e_2\ e_3) : t, W} \\ \text{if } t \geq t_2, t \geq t_3, W(x) \geq W_1(x) \sqcap (W_2(x) \sqcup W_3(x)) \text{ for all } x$$

$$\frac{((f : t), \Gamma) \vdash_{sa} e : t, (b, W)}{\Gamma \vdash_{sa} (\mathsf{rec}\ f\ e) : t', W} \text{ if } t' \geq t$$

Figure 5: The result of inlining the subsumption rule.

4.1 Getting a Constraint-Based Inference System

The first step will be to "inline" the subsumption rule into (some of) the other rules. The result is depicted in Fig. 5.

Fact 4.1 A judgement is derivable in the system in Fig. 4 iff it is derivable in the system in Fig. 5.

PROOF: The "if"-part is an easy induction in the proof tree. For the "only if"-part, we need that if $\Gamma \vdash_{sa} e : t, W$ using Fig. 5 and $t \leq t', W \leq W'$ then also $\Gamma \vdash_{sa} e : t', W'$ using Fig. 5. This follows from a more general fact, which easily can be proved by induction in the proof tree: if $\Gamma_{sa} e : t, W$ using Fig. 5 and $\Gamma' \leq \Gamma$ (pointwise), $t \leq t'$ and $W \leq W'$ then $\Gamma' \vdash_{sa} e : t', W'$ using Fig. 5. $\Box$

It will be convenient to reformulate the system in Fig. 5, using the $t[\vec{b}^+, \vec{b}^-]$-notation thus making annotations on arrows more explicit. This is done[5] in Fig. 6; it is immediate to verify that this system is equivalent to the one in Fig. 5. A remark about covariant/contravariant position: the turnstile $\vdash$ acts like an $\rightarrow$, so if $t = \Gamma(x)$ then covariant positions in t will be considered as appearing contravariantly in the judgement, and vice versa. On the other hand, something appearing in the range of W is considered as being in covariant position in the judgement. We shall consistently use the convention that a superscript + (e.g. as in $\vec{b}_1^+$) indicates something which appears on an arrow in *covariant* position in the judgement; whereas a superscript – (e.g. as in $\vec{b}_2^-$) indicates something which appears on an arrow in *contravariant* position in the judgement. It is important to notice that this *can* be done consistently, i.e. that polarity is always the same in the premise as in the conclusion.

We shall now introduce *open strictness types:* this is similar to strictness types except that the arrows are annotated not by 0's and 1's, but by a certain kind of variables to be called *strictness variables*. An open strictness type t, together with a mapping from the strictness variables in t into $\{0, 1\}$, in the natural way determines a strictness type.

Notice that in Fig. 6 the $\vec{b}^+$'s and the $\vec{b}^-$'s really are *meta variables*, ranging over 0 and 1. By making them range over *strictness variables* we shall obtain a type inference system, depicted in Fig. 7, with judgements of form $\Gamma \vdash_{sa} e : t, W, C$. Here Γ is an environment (represented as a list) mapping (program) variables into open strictness types; t is an open strictness type; W is a mapping (represented as a list) from program variables into strictness variables; and C is a set of *constraints* among strictness variables.

[5]For space reasons we shall employ the convention that e.g. "$\Gamma \vdash_{sa} e_1 : t_1, W_1$ and $e_2 : t_2, W_2$" means "$\Gamma \vdash_{sa} e_1 : t_1, W_1$ and $\Gamma \vdash_{sa} e_2 : t_2, W_2$".

4.2 Solving the Constraints

The inference system in Fig. 7 does in the obvious way give rise to a deterministic algorithm, provided

1. we are able to find the underlying types (i.e. without strictness annotations) of all expressions;
2. we are able to solve the constraints generated.

As already mentioned we shall assume that 1 has been done in advance; so our aim will be to give an algorithm for solving the constraints generated by the system. Our approach will be to show how to *normalize* this system, thus showing that the solutions have a particular form. First some preliminaries:

Strictness Expressions

Strictness expressions will be built from strictness variables, 0, 1, $\sqcap$ and $\sqcup$. Thus a strictness expression s in the natural way gives rise to a monotone function g with domain the free variables of s (and all monotone functions on finite domains can be represented by strictness expressions).

Extended Constraints

The constraints met in Fig. 7 are of form $\vec{b_1} \geq g(\vec{b_2})$ or of form $\vec{b_1} = g(\vec{b_2})$, with g being a monotone function. We introduce a new kind of constraints, which besides using $\geq$ and $=$ also use a special symbol $\gg$. Intuitively, $b_1 \gg g(b_2)$ "lies between" $b_1 = g(b_2)$ and $b_1 \geq g(b_2)$. More precisely, we have:

Definition 4.2 Let N be an extended constraint system (i.e. possibly containing $\gg$). Let ϕ be a mapping from the strictness variables of N into $\{0, 1\}$. We say that ϕ is a *strong solution* to N iff ϕ is a solution to the system resulting from replacing all $\gg$'s in N by $=$; and we say that ϕ is a *weak sohtion* to N iff ϕ is a solution to the system resulting from replacing all $\gg$'s in N by $\geq$.

$$\Gamma[\vec{b}^-, \vec{b}^+] \vdash_{sa} c : Ct(c)[\vec{b}_1^+, ()], \vec{b}_2^+$$
$$\text{if } \vec{b}_1^+ \geq \vec{0}, \vec{b}_2^+ \geq \vec{1}$$

$$(\Gamma_1[\vec{b}_1^-, \vec{b}_1^+], (x : t[\vec{b}_2^-, \vec{b}_2^+]), \Gamma_2[\vec{b}_3^-, \vec{b}_3^+]) \vdash_{sa} x : t[\vec{b}_4^+, \vec{b}_4^-], (\vec{b}_5^+, b_6^+, \vec{b}_7^+)$$
$$\text{if } \vec{b}_4^+ \geq \vec{b}_2^-, \vec{b}_2^+ \geq \vec{b}_4^-, \vec{b}_5^+ \geq 1, b_6^+ \geq 0, \vec{b}_7^+ \geq 1$$

$$\frac{((x : t_1[\vec{b}_1^-, \vec{b}_1^+]), \Gamma[\vec{b}^-, \vec{b}^+]) \vdash_{sa} e : t[\vec{b}_2^+, \vec{b}_2^-], (b_3^+, \vec{b}_4^+)}{\Gamma[\vec{b}^-, \vec{b}^+] \vdash_{sa} \lambda x.e : t_1[\vec{b}_1^-, \vec{b}_1^+] \rightarrow_{b_3^+} t[\vec{b}_2^+, \vec{b}_2^-], \vec{b}_4^+}$$

$$\frac{\Gamma[\vec{b}^-, \vec{b}^+] \vdash_{sa} e_1 : t_2[\vec{b}_2^-, \vec{b}_2^+] \rightarrow_{b_3^+} t_1[\vec{b}_4^+, \vec{b}_4^-], \vec{b}_5^+ \text{ and } e_2 : t_2[\vec{b}_6^+, \vec{b}_6^-], \vec{b}_7^+}{\begin{array}{c}\Gamma[\vec{b}^-, \vec{b}^+] \vdash_{sa} e_1 e_2 : t_1[\vec{b}_4^+, \vec{b}_4^-], \vec{b}_8^+ \\ \text{if } \vec{b}_6^+ = \vec{b}_2^-, \vec{b}_2^+ = \vec{b}_6^-, \vec{b}_8^+ \geq \vec{b}_5^+ \sqcap (b_3^+ \sqcup \vec{b}_7^+)\end{array}}$$

$$\frac{\Gamma[\vec{b}^-, \vec{b}^+] \vdash_{sa} e_1 : \mathsf{Bool}, \vec{b}_3^+ \text{ and } e_2 : t[\vec{b}_4^+, \vec{b}_4^-], \vec{b}_5^+ \text{ and } e_3 : t[\vec{b}_6^+, \vec{b}_6^-], \vec{b}_7^+}{\begin{array}{c}\Gamma \vdash_{sa} (\mathsf{if}\ e_1\ e_2\ e_3) : t[\vec{b}_8^+, \vec{b}_8^-], \vec{b}_9^+ \\ \text{if } \vec{b}_8^+ \geq \vec{b}_4^+, \vec{b}_4^- \geq \vec{b}_8^-, \vec{b}_8^+ \geq \vec{b}_6^+, \vec{b}_6^- \geq \vec{b}_8^-, \vec{b}_9^+ \geq \vec{b}_3^+ \sqcap (\vec{b}_5^+ \sqcup \vec{b}_7^+)\end{array}}$$

$$\frac{((f : t[\vec{b}_1^-, \vec{b}_1^+]), \Gamma[\vec{b}^-, \vec{b}^+]) \vdash_{sa} e : t[\vec{b}_2^+, \vec{b}_2^-], (b_3^+, \vec{b}_4^+)}{\begin{array}{c}\Gamma[\vec{b}^-, \vec{b}^+] \vdash_{sa} (\mathsf{rec}\ f\ e) : t[\vec{b}_5^+, \vec{b}_5^-], \vec{b}_4^+ \\ \text{if } \vec{b}_5^+ \geq \vec{b}_2^+, \vec{b}_2^- \geq \vec{b}_5^-, \vec{b}_2^+ = \vec{b}_1^-, \vec{b}_2^- = \vec{b}_1^+\end{array}}$$

Figure 6: Annotations on arrows made explicit.

$$\Gamma[\vec{b}^-, \vec{b}^+] \vdash_{sa} c : Ct(c)[\vec{b}_1^+, ()], \vec{b}_2^+, \\ \{\vec{b}_1^+ \geq \vec{0}, \vec{b}_2^+ \geq \vec{1}\}$$

$$(\Gamma_1[\vec{b}_1^-, \vec{b}_1^+], (x : t[\vec{b}_2^-, \vec{b}_2^+]), \Gamma_2[\vec{b}_3^-, \vec{b}_3^+]) \vdash_{sa} x : t[\vec{b}_4^+, \vec{b}_4^-], (\vec{b}_5^+, b_6^+, \vec{b}_7^+), \\ \{\vec{b}_4^+ \geq \vec{b}_2^-, \vec{b}_2^+ \geq \vec{b}_4^-, \vec{b}_5^+ \geq 1, b_6^+ \geq 0, \vec{b}_7^+ \geq 1\}$$

$$\frac{((x : t_1[\vec{b}_1^-, \vec{b}_1^+]), \Gamma[\vec{b}^-, \vec{b}^+]) \vdash_{sa} e : t[\vec{b}_2^+, \vec{b}_2^-], (b_3^+, \vec{b}_4^+), C}{\Gamma[\vec{b}^-, \vec{b}^+] \vdash_{sa} \lambda x.e : t_1[\vec{b}_1^-, \vec{b}_1^+] \rightarrow_{b_3^+} t[\vec{b}_2^+, \vec{b}_2^-], \vec{b}_4^+, C}$$

$$\frac{\Gamma[\vec{b}^-, \vec{b}^+] \vdash_{sa} e_1 : t_2[\vec{b}_2^-, \vec{b}_2^+] \rightarrow_{b_3^+} t_1[\vec{b}_4^+, \vec{b}_4^-], \vec{b}_5^+, C_1 \text{ and } e_2 : t_2[\vec{b}_6^+, \vec{b}_6^-], \vec{b}_7^+, C_2}{\Gamma[\vec{b}^-, \vec{b}^+] \vdash_{sa} e_1 e_2 : t_1[\vec{b}_4^+, \vec{b}_4^-], \vec{b}_8^+, C_1 \cup C_2 \cup C \\ \text{where } C = \{\vec{b}_6^+ = \vec{b}_2^-, \vec{b}_2^+ = \vec{b}_6^-, \vec{b}_8^+ \geq \vec{b}_5^+ \sqcap (b_3^+ \sqcup \vec{b}_7^+)\}}$$

$$\frac{\Gamma[\vec{b}^-, \vec{b}^+] \vdash_{sa} e_1 : \mathsf{Bool}, \vec{b}_3^+, C_1 \text{ and } e_2 : t[\vec{b}_4^+, \vec{b}_4^-], \vec{b}_5^+, C_2 \text{ and } e_3 : t[\vec{b}_6^+, \vec{b}_6^-], \vec{b}_7^+, C_3}{\Gamma[\vec{b}^-, \vec{b}^+] \vdash_{sa} (\mathsf{if}\ e_1\ e_2\ e_3) : t[\vec{b}_8^+, \vec{b}_8^-], \vec{b}_9^+, C_1 \cup C_2 \cup C_3 \cup C \\ \text{where } C = \{\vec{b}_8^+ \geq \vec{b}_4^+, \vec{b}_4^- \geq \vec{b}_8^-, \vec{b}_8^+ \geq \vec{b}_6^+, \vec{b}_6^- \geq \vec{b}_8^-, \vec{b}_9^+ \geq \vec{b}_3^+ \sqcap (\vec{b}_5^+ \sqcup \vec{b}_7^+)\}}$$

$$\frac{((f : t[\vec{b}_1^-, \vec{b}_1^+]), \Gamma[\vec{b}^-, \vec{b}^+]) \vdash_{sa} e : t[\vec{b}_2^+, \vec{b}_2^-], (b_3^+, \vec{b}_4^+), C}{\Gamma[\vec{b}^-, \vec{b}^+] \vdash_{sa} (\mathsf{rec}\ f\ e) : t[\vec{b}_5^+, \vec{b}_5^-], \vec{b}_4^+, C \cup C' \\ \text{where } C' = \{\vec{b}_5^+ \geq \vec{b}_2^+, \vec{b}_2^- \geq \vec{b}_5^-, \vec{b}_2^+ = \vec{b}_1^-, \vec{b}_2^- = \vec{b}_1^+\}}$$

Figure 7: An inference system collecting constraints.

Let C be a constraint system without $\gg$, and let N be a constraint system possibly containing $\gg$. We say that $C \sim N$ iff all strong solutions to N are solutions to C, and all solutions to C are weak solutions to N.

Normalizing Constraints

Given an expression e, we by means of the system in Fig. 7 are able to produce a proof tree for $() \vdash_{sa} e : t, (), C$. The strictness variables occurring in C can be divided into two groups: those occurring in t, to be denoted $\vec{b}_1^+$ and $\vec{b}_1^-$, and those which do not occur in t (but further up in the proof tree), to be denoted $\vec{b}_0$ (we do not distinguish between covariant and contravariant here). It turns out that we are able to produce an extended constraint system N with the following property:

- $C \sim N$;
- N is of form $\{\vec{b}_1^+ \geq \vec{s}_1; \vec{b}_0 \gg \vec{s}_0\}$, with $\vec{s}_0$ and $\vec{s}_1$ being strictness expressions whose (free) variables belong to $\vec{b}_1^-$.

Thus each choice of the contravariant positions in t gives rise to a *least* annotation of the covariant positions (but all greater annotations are also solutions).

Example 4.3 The expression $\lambda f.\lambda x.f(f(x))$ will have type

$$(\mathsf{int}\rightarrow_{b_1^-}\mathsf{Int})\rightarrow_{b_{11}^+} (\mathsf{int}\rightarrow_{b_{12}^+}\mathsf{Int})$$

with constraint $b_{12}^+ \geq b_1^-$. So the minimal types of this expression are $(\mathsf{int}\rightarrow_0\mathsf{Int})\rightarrow_0(\mathsf{int}\rightarrow_0\mathsf{Int})$ and $(\mathsf{int}\rightarrow_1\mathsf{Int})\rightarrow_0(\mathsf{int}\rightarrow_1\mathsf{Int})$. □

The reason for also being interested in the value of $\vec{b}_0$ is that the strictness types of the subexpressions may be useful, e.g. to produce the translation in Sect. 5. We should of course use the least such value.

Example 4.4 Consider the expression e_1e_2, where $e_1 = \lambda f.\lambda x.f(f(x))$ and $e_2 = (\lambda y.y)$. e_1 has type $(\mathsf{int}\rightarrow_{b_{01}^-}\mathsf{Int})\rightarrow_{b_{01}^+}(\mathsf{int}\rightarrow_{b_1^+}\mathsf{Int})$, e_2 has type $\mathsf{int}\rightarrow_{b_{02}^+}\mathsf{Int}$

and e_1e_2 has type $\mathsf{int}\rightarrow_{b_1^+}\mathsf{Int}$), where the constraints look like $b_{02}^+ = b_{01}^-$ and $b_1^+ \geq b_{01}^-$ This can be normalized into $b_1^+ \geq 0, b_{02}^+ \gg 0$ and $b_{01}^- \gg 0$. So one should assign e_1 type $(\mathsf{int}\rightarrow_0\mathsf{Int})\rightarrow_0(\mathsf{int}\rightarrow_0\mathsf{Int})$, that is assume that f has type $\mathsf{int}\rightarrow_0\mathsf{Int}$. $\Box$

In order to achieve our goal we for each rule in Fig. 7 of form

$$\frac{\ldots \Gamma_i \vdash_{sa} e_i \ : \ t_i, W_i, C_i \ldots}{\Gamma \vdash_{sa} e \ : \ t, W, C_1 \cup \ldots \cup C_n \cup C}$$

must proceed as follows: assume that there for each i exists an extended constraint system N_i such that

- $C_i \sim N_i$;
- N_i is of form $\{\vec{b}_{i1}^+ \geq \vec{s}_{i1}, \vec{b}_{i0} \gg \vec{s}_{i0}\}$ where $(\vec{b}_{i1}^+, \vec{b}_{i1}^-)$ are the strictness variables occurring in Γ_i, t_i or W_i, where $\vec{b}_{i0}$ are the remaining strictness variables in C_i, and where the free variables of the strictness expressions $\vec{s}_{i1}$ and $\vec{s}_{i0}$ belong to $\vec{b}_{i1}^-$.

Then we must be able to construct N (i.e. do a *normalization*) such that

- $C_1 \cup \cdots C_n \cup C \sim N$;
- N is of form $\{\vec{b}_1^+ \geq \vec{s}_1, \vec{b}_0 \gg \vec{s}_0\}$ where $(\vec{b}_1^+, \vec{b}_1^-)$ are the strictness variables occurring in Γ, t or W, where $\vec{b}_0$ are the remaining strictness variables in C, and where the free variables of the strictness expressions $\vec{s}_1$ and $\vec{s}_0$ belong to $\vec{b}_1^-$.

Before embarking on the normalization process, it will be convenient to describe some of the tools to be used.

Some Transformation Rules

Let N and N' be extended constraint systems. We say that it is correct to transform N into N' if all strong solutions to N' also are strong solutions

to N, and if all weak solutions to N also are weak solutions to N'. If it is correct to transform N into N', we clearly have that if $C \sim N$ then also $C \sim N'$. We now list some correct transformations:

Fact 4.5 Suppose N contains the constraints $\vec{b} \geq \vec{s}_1, \vec{b} \geq \vec{s}_2$. Then these can be replaced by the constraint $\vec{b} \geq \vec{s}_1 \sqcup \vec{s}_2$.

Fact 4.6 Suppose N contains the constraint $\vec{b} \geq \vec{s}$ or the constraint $\vec{b} = \vec{s}$. Then this can be replaced by $\vec{b} \gg \vec{s}$.

Fact 4.7 Suppose N contains the constraints $\vec{b}_1 \geq g_1(\vec{b}_2)$ and $\vec{b}_2 \gg g_2(\vec{b}_3)$. Then the former constraint can be replaced by the constraint $\vec{b}_1 \geq g_1(g_2(\vec{b}_3))$, yielding a new constraint system N'.

(In other words, if we have the constraints $\vec{b} \geq \vec{s}$ and $\vec{b}_i \gg \vec{s}_i$ it is safe to replace the former constraint by $\vec{b} \geq \vec{s}[\vec{s}_i/\vec{b}_i]$.)

PROOF: Let a strong solution to N' be given. Wrt. this solution, we have $\vec{b}_2 = g_2(\vec{b}_3)$, $\vec{b}_1 \geq g_1(g_2(\vec{b}_3))$ and hence also $\vec{b}_1 \geq g_1(\vec{b}_2)$ – thus this solution is also a strong solution to N.

Now let a weak solution to N be given. Wrt. this solution, we have $\vec{b}_1 \geq g_1(\vec{b}_2)$ and $\vec{b}_2 \geq g_2(\vec{b}_3)$. Due to the monotonicity of g_1, we also have $\vec{b}_1 \geq g_1(g_2(\vec{b}_3))$ showing that this solution is also a weak solution to N'. □

Fact 4.8 Suppose N contains the constraints $\vec{b}_1 \gg g_1(\vec{b}_2)$ and $\vec{b}_2 \gg g_2(\vec{b}_3)$. Then the former constraint can be replaced by the constraint $\vec{b}_1 \gg g_1(g_2(\vec{b}_3))$, yielding a new constraint system N'.

(In other words, if we have the constraints $\vec{b} \gg \vec{s}$ and $\vec{b}_i \gg \vec{s}_i$ it is safe to replace the former constraint by $\vec{b} \gg \vec{s}[\vec{s}_i/\vec{b}_i]$.)

PROOF: Let a strong solution to N' be given. Wrt. this solution, we have $\vec{b}_2 = g_2(\vec{b}_3)$, $\vec{b}_1 = g_1(g_2(\vec{b}_3))$ and hence also $\vec{b}_1 = g_1(\vec{b}_2)$ – thus this solution is also a strong solution to N.

Now let a weak solution to N be given. Wrt. this solution, we have $\vec{b}_1 \geq g_1(\vec{b}_2)$ and $\vec{b}_2 \geq g_2(\vec{b}_3)$. Due to the monotonicity of g_1, we also have $\vec{b}_1 \geq g_1(g_2(\vec{b}_3))$ showing that this solution is also a weak solution to N'. □

Fact 4.9 Suppose N contains the constraint $\vec{b}_1 \geq g(\vec{b}_1, \vec{b}_2)$. Then this can be replaced by the constraint $\vec{b}_1 \gg g'(\vec{b}_2)$ (yieldig N'), where

$$g'(\vec{b}_2) = \sqcup_k h^k(\vec{0}) \text{ with } h(\vec{b}) = g(\vec{b}, \vec{b}_2)$$

(this just amounts to Tarski's theorem – notice that it will actually suffice with $|\vec{b}_1|$ iterations).

PROOF: First suppose that we have a strong solution to N', i.e. $\vec{b}_1 = g'(\vec{b}_2)$. Since $\vec{b}_1 = \sqcup_k h^k(\vec{0})$, standard reasoning on the monotone and hence (as finite lattices) continuous function h tells us that $\vec{b}_1 = h(\vec{b}_1)$, i.e. $\vec{b}_1 = g(\vec{b}_1, \vec{b}_2)$. This shows that we have a strong solution to N.

Now suppose that we have a weak solution to N, i.e. $\vec{b}_1 \geq g(\vec{b}_1, \vec{b}_2)$. In order to show that this also is a weak solution to N', we must show that $\vec{b}_1 \geq g'(\vec{b}_2)$. This can be done by showing that $\vec{b}_1 \geq \vec{b}$ implies $\vec{b}_1 \geq h(\vec{b})$. But if $\vec{b}_1 \geq \vec{b}$ we have $\vec{b}_1 \geq g(\vec{b}_1, \vec{b}_2) \geq g(\vec{b}, \vec{b}_2) = h(\vec{b})$.

It is easy to see that g' is monotone. □

4.3 The Normalization Process

We shall examine the various constructs: for constants and variables the normalization process is trivial, as the constraints generated are of the required form. Neither does the rule for abstractions cause any troubles, since no new constraints are generated and since a strictness variable appears in the premise of the rule iff it appears in the conclusion (and with the same polarity). Now let us focus upon the remaining constructs.

Normalizing the Rule for Application

Recall the rule

$$\frac{\Gamma[\vec{b}^-, \vec{b}^+] \vdash_{sa} e_1 \;\; t_2[\vec{b}_2^-, \vec{b}_2^+] \rightarrow_{b_3^+} t_1[\vec{b}_4^+, \vec{b}_4^-], \vec{b}_5^+, C_1 \text{ and } e_2 \;:\; t_2[\vec{b}_6^+, \vec{b}_6^-], \vec{b}_7^+, C_2}{\Gamma[\vec{b}^-, \vec{b}^+] \vdash_{sa} e_1 e_2 \;:\; t_1[\vec{b}_4^+, \vec{b}_4^-], \vec{b}_8^+, C_1 \cup C_2 \cup C}$$

where $C = \{\vec{b}_6^+ = \vec{b}_2^-, \vec{b}_2^+ = \vec{b}_6^-, \vec{b}_8^+ \geq \vec{b}_5^+ \sqcap (\vec{b}_3^+ \sqcup \vec{b}_7^+)\}$.

Let $\vec{b}_0$ be the "extra" strictness variables of C_1 and $\vec{b}_1$ the extra strictness variables of C_2. Then we are entitled to assume that there exist N_1, N_2 with $C_1 \sim N_1, C_2 \sim N_2$ such that N_1 takes the form

$$\vec{b}^+ \geq \vec{s}_a \;\; \vec{b}_2^+ \geq \vec{s}_2 \;\; b_3^+ \geq s_3 \;\; \vec{b}_4^+ \geq \vec{s}_4 \;\; \vec{b}_5^+ \geq \vec{s}_5 \;\; \vec{b}_0 \gg \vec{s}_0$$

(where the free variables of the strictness expressions above belong to $\{\vec{b}^-, \vec{b}_2^-, \vec{b}_4^-\}$) and such that N_2 takes the form

$$\vec{b}^+ \geq \vec{s}_b \;\; \vec{b}_6^+ \geq \vec{s}_6 \;\; \vec{b}_7^+ \geq \vec{s}_7 \;\; \vec{b}_1 \gg \vec{s}_1$$

(where the free variables of the strictness expressions above belong to $\{\vec{b}^-, \vec{b}_6^-\}$.)

Clearly $C_1 \cup C_2 \cup C \sim N_1 \cup N_2 \cup C$. We shall now manipulate $N_1 \cup N_2 \cup C$ (by means of correct transformations) with the aim of getting something of the desired form.

The first step is to use Fact 4.5 to replace the two inequalities for $\vec{b}^+$ by one, and at the same time exploit that $\vec{b}_2^+ = \vec{b}_6^-$ and $\vec{b}_6^+ = \vec{b}_2^-$. As a result, we arrive at

$$\begin{array}{llll} \vec{b}^+ \geq \vec{s}_a \sqcup \vec{s}_b & \vec{b}_6^- \geq \vec{s}_2 & \vec{b}_3^+ \geq s_3 & \vec{b}_4^+ \geq \vec{s}_4 \\ \vec{b}_5^+ \geq \vec{s}_5 & \vec{b}_0 \gg \vec{s}_0 & \vec{b}_2^- \geq \vec{s}_6 & \vec{b}_7^+ \geq \vec{s}_7 \\ \vec{b}_1 \gg \vec{s}_1 & \vec{b}_6^+ = \vec{b}_2^- & \vec{b}_2^+ = \vec{b}_6^- & \vec{b}_8^+ \geq \vec{b}_5^+ \sqcap (b_3^+ \sqcup \vec{b}_7^+) \end{array}$$

We now focus upon the pair of constraints

$$(\vec{b}_6^-, \vec{b}_2^+) \geq (\vec{s}_2, \vec{s}_6)$$

According to Fact 4.9, these can be replaced by the constraints

$$(\vec{b}_6^-, \vec{b}_2^+) \gg (\vec{S}_2, \vec{S}_6)$$

where $(\vec{S}_2, \vec{S}_6)$ is given as the "limit" of the chain with elements $(\vec{s}_{2n}, \vec{s}_{6n})$ given by

$$\begin{array}{rcl}(\vec{s}_{20}, \vec{s}_{60}) & = & \vec{0} \\ (\vec{s}_{2(n+1)}, \vec{s}_{6(n+1)}) & = & (\vec{s}_2, \vec{s}_6)[(\vec{s}_{2n}, \vec{s}_{6n})/(\vec{b}_6^-, \vec{b}_2^-)]\end{array}$$

This limit can be found as the k'th element, where $k = |(\vec{b}_2^-, \vec{b}_6^-)|$.

As $\vec{s}_2$ does not contain $\vec{b}_6^-$ and $\vec{s}_6$ does not contain $\vec{b}_2^-$, the above can be simplified into

$$\begin{array}{ll}\vec{s}_{20} = \vec{0} & \vec{s}_{2(n+1)} = \vec{s}_2[\vec{s}_{6n}/\vec{b}_2^-] \\ \vec{s}_{20} = \vec{0} & \vec{s}_{6(n+1)} = \vec{s}_6[\vec{s}_{2n}/\vec{b}_6^-]\end{array}$$

Our next step is to substitute in the new constraints for $\vec{b}_2^-$ and $\vec{b}_6^-$, using Fact 4.7 and Fact 4.8. We arrive at (after also having used Fact 4.6)

$$\begin{array}{lll}\vec{b}^+ \geq \vec{s}_a[\vec{S}_6/\vec{b}_2^-] \sqcup \vec{s}_b[\vec{S}_2/\vec{b}_6^-] & \vec{b}_6^- \gg \vec{S}_2 & \vec{b}_3^+ \gg s_3[\vec{S}_6/\vec{b}_2^-] \\ \vec{b}_4^+ \geq \vec{s}_4[\vec{S}_6/\vec{b}_2^-] & \vec{b}_5^+ \gg \vec{s}_5[\vec{S}_6/\vec{b}_2^-] & \vec{b}_0 \gg \vec{s}_0[\vec{S}_6/\vec{b}_2^-] \\ \vec{b}_2^- \gg \vec{S}_6 & \vec{b}_7^+ \gg \vec{s}_7[\vec{S}_2/\vec{b}_6^-] & \vec{b}_1 \gg \vec{s}_1[\vec{S}_2/\vec{b}_6^-] \\ \vec{b}_6^+ \gg \vec{S}_6 & \vec{b}_2^+ \gg \vec{S}_2 & \vec{b}_8^+ \geq \vec{b}_5^+ \sqcap (\vec{b}_3^+ \sqcup \vec{b}_7^+)\end{array}$$

Finally we use Fact 4.7 on the constraint for $\vec{b}_8^+$, arriving at

$$\begin{array}{lll}\vec{b}^+ \geq \vec{s}_a[\vec{S}_6/\vec{b}_2^-] \sqcup \vec{s}_b[\vec{S}_2/\vec{b}_6^-] & \vec{b}_6^- \gg \vec{S}_2 & b_3^+ \gg s_3[\vec{S}_6/\vec{b}_2^-] \\ b_4^+ \geq \vec{s}_4[\vec{S}_6/\vec{b}_2^-] & \vec{b}_5^+ \gg \vec{s}_5[\vec{S}_6/\vec{b}_2^-] & \vec{b}_0 \gg \vec{s}_0[\vec{S}_6/\vec{b}_2^-] \\ \vec{b}_2^- \gg \vec{S}_6 & \vec{b}_7^+ \gg \vec{s}_7[\vec{S}_2/\vec{b}_6^-] & \vec{b}_1 \gg \vec{s}_1[\vec{S}_6/\vec{b}_2^-] \\ \vec{b}_2^- \gg \vec{S}_6 & \vec{b}_2^+ \gg \vec{S}_2 & \\ \vec{b}_8^+ \geq \vec{s}_5[\vec{S}_6/\vec{b}_2^-] \sqcap (s_3[\vec{S}_2/\vec{b}_6^-] \sqcup \vec{s}_7[\vec{S}_6/\vec{b}_2^-]) & & \end{array}$$

This is of the desired form, as it is quite easy to check that the only strictness variables occurring in the expressions on the right hand sides are those occurring in $\vec{b}^-$ and in $\vec{b}_4^-$.

Normalizing the Rule for Conditionals

Recall the rule

$$\frac{\Gamma[\vec{b}^-,\vec{b}^+] \vdash_{sa} e_1 \ : \ \mathsf{Bool}, \vec{b}_3^+, C_1 \text{ and } e_2 \ : \ t[\vec{b}_4^+,\vec{b}_4^-],\vec{b}_5^+, C_2 \text{ and } e_3 \ : \ t[\vec{b}_6^-,\vec{b}_6^-],\vec{b}_7^+, C_3}{\Gamma[\vec{b}^-,\vec{b}^+] \vdash_{sa} \mathsf{if}\ e_1\ e_2\ e_3 \ : \ t[\vec{b}_8^+,\vec{b}_8^-],\vec{b}_9^+, C_1 \cup C_2 \cup C_3 \cup C}$$

where $C = \{\vec{b}_8^+ \geq \vec{b}_4^+, \vec{b}_8^+ \geq \vec{b}_6^+, \vec{b}_4^- \geq \vec{b}_8^-, \vec{b}_6^- \geq \vec{b}_8^-, \vec{b}_9^+ \geq \vec{b}_3^+ \sqcap (\vec{b}_5^+ \sqcup \vec{b}_7^+)\}$.

Let $\vec{b}_0$ be the "extra" strictness variables of C_1, $\vec{b}_1$ the extra strictness variables of C_2 and $\vec{b}_2$ the extra strictness variables of C_3. Then we are entitled to assume that there exist N_1, N_2, N_3 with $C_1 \sim N_1, C_2 \sim N_2, C_3 \sim N_3$ such that N_1 takes the form

$$\vec{b}^+ \geq \vec{s}_a \ \ \vec{b}_3^+ \geq \vec{s}_3 \ \ \vec{b}_0 \gg \vec{s}_0$$

(where the free variables of the strictness expressions above belong to $\vec{b}^-$) and such that N_2 takes the form

$$\vec{b}^+ \geq \vec{s}_b \ \ \vec{b}_4^+ \geq \vec{s}_4 \ \ \vec{b}_5^+ \geq \vec{s}_5 \ \ \vec{b}_1 \gg \vec{s}_1$$

(where the free variables of the strictness expressions above belong to ($\vec{b}^-$, $\vec{b}_4^-$)) and such that N_3 takes the form

$$\vec{b}^+ \geq \vec{s}_c \ \ \vec{b}_6^+ \geq \vec{s}_6 \ \ \vec{b}_7^+ \geq \vec{s}_7 \ \ \vec{b}_2 \gg \vec{s}_2$$

(where the free variables of the strictness expressions above belong to ($\vec{b}^-$, $\vec{b}_6^-$)).

Clearly $C_1 \cup C_2 \cup C_3 \cup C \sim N_1 \cup N_2 \cup N_3 \cup C$. we shall now manipulate $N_1 \cup N_2 \cup N_3 \cup C$ (by means of correct transformations) with the aim of getting something of the desired form.

The first step is to use Fact 4.6 (i.e. replace $\geq$ by $\gg$) on the inequalities $\vec{b}_4^- \geq \vec{b}_8^-$ and $\vec{b}_6^- \geq \vec{b}_8^-$, afterwards use Fact 4.7 and Fact 4.8 to substitute $\vec{b}_4^-$ ($\vec{b}_6^-$) by $\vec{b}_8^-$. By also using Fact 4.5 to replace the two inequalities for $\vec{b}_8^+$ by one and to replace the three inequalities for $\vec{b}^+$ by one, we arrive at

$$
\begin{array}{lll}
\vec{b}^+ \geq \vec{s}_a \sqcup \vec{s}_b[\vec{b}_8^-/\vec{b}_4^-] \sqcup \vec{s}_c[\vec{b}_8^-/\vec{b}_6^-] & \vec{b}_3^+ \geq \vec{s}_3 & \vec{b}_0 \gg \vec{s}_0 \\
\vec{b}_4^+ \geq \vec{s}_4[\vec{b}_8^-/\vec{b}_4^-] & \vec{b}_5^+ \geq \vec{s}_5[\vec{b}_8^-/\vec{b}_4^-] & \vec{b}_1 \gg \vec{s}_1[\vec{b}_8^-/\vec{b}_4^-] \\
\vec{b}_6^+ \geq \vec{s}_6[\vec{b}_8^-/\vec{b}_6^-] & \vec{b}_7^+ \geq \vec{s}_7[\vec{b}_8^-/\vec{b}_6^-] & \vec{b}_2 \gg \vec{s}_2[\vec{b}_8^-/\vec{b}_6^-] \\
\vec{b}_8^+ \geq \vec{b}_4^+ \sqcup \vec{b}_6^+ & \vec{b}_4^- \gg \vec{b}_8^- & \vec{b}_6^- \gg \vec{b}_8^- \\
\vec{b}_9^+ \geq \vec{b}_3^+ \sqcap (\vec{b}_5^+ \sqcup \vec{b}_7^+) & &
\end{array}
$$

Next we replace some $\geq$ by $\gg$ (Fact 4.6), enabling us to use Fact 4.7 on the constraints for $\vec{b}_8^+$ and $\vec{b}_9^+$. We arrive at

$$
\begin{array}{lll}
\vec{b}^+ \geq \vec{s}_a \sqcup \vec{s}_b[\vec{b}_8^-/\vec{b}_4^-] \sqcup \vec{s}_c[\vec{b}_8^-/\vec{b}_6^-] & \vec{b}_3^+ \gg \vec{s}_3 & \vec{b}_0 \gg \vec{s}_0 \\
\vec{b}_4^+ \gg \vec{s}_4[\vec{b}_8^-/\vec{b}_4^-] & \vec{b}_5^+ \gg \vec{s}_5[\vec{b}_8^-/\vec{b}_4^-] & \vec{b}_1 \gg \vec{s}_1[\vec{b}_8^-/\vec{b}_4^-] \\
\vec{b}_6^+ \gg \vec{s}_6[\vec{b}_8^-/\vec{b}_6^-] & \vec{b}_7^+ \gg \vec{s}_7[\vec{b}_8^-/\vec{b}_6^-] & \vec{b}_2 \gg \vec{s}_2[\vec{b}_8^-/\vec{b}_6^-] \\
\vec{b}_8^+ \geq \vec{s}_4[\vec{b}_8^-/\vec{b}_4^-] \sqcup [\vec{b}_8^-/\vec{b}_6^-] & \vec{b}_4^- \gg \vec{b}_8^- & \vec{b}_6^- \gg \vec{b}_8^- \\
\vec{b}_9^+ \geq \vec{s}_3 \sqcap (\vec{s}_5[\vec{b}_8^-/\vec{b}_4^-] \sqcap \vec{s}_7[\vec{b}_8^-/\vec{b}_6^-]) & &
\end{array}
$$

This is of the desired form, as it is quite easy to check that the only strictness variables occurring in the expressions on the right hand sides are those occurring in $\vec{b}^-$ and in $\vec{b}_8^-$.

Normalizing the Rule for Recursion

Recall the rule

$$
\frac{((f \ : \ t[\vec{b}_1^-, \vec{b}_1^+]), \Gamma[\vec{b}^-, \vec{b}^+]) \vdash_{sa} e \ : \ t[\vec{b}_2^+, \vec{b}_2^-], (b_3^+, \vec{b}_4^+), C}{\Gamma[\vec{b}^-, \vec{b}^+] \vdash_{sa} \mathsf{rec}\ f\ e \ : \ t[\vec{b}_5^+, \vec{b}_5^-], \vec{b}_4^+, C \cup C'}
$$

where $C' = \{\vec{b}_1^- = \vec{b}_2^+,\ \vec{b}_2^- = \vec{b}_1^+,\ \vec{b}_5^+ \geq \vec{b}_2^+,\ \vec{b}_2^- \geq \vec{b}_5^-\}$.

Let b_0 be the "extra" strictness variables of C. We are entitled to assume that there exist N with $C \sim N$ such that N takes the form

$$
\vec{b}^+ \geq \vec{s}\ \ \vec{b}_1^+ \geq \vec{s}_1\ \ \vec{b}_2^+ \geq \vec{s}_2\ \ \vec{b}_3^+ \geq \vec{s}_3\ \ \vec{b}_4^+ \geq \vec{s}_4\ \ \vec{b}_0 \gg \vec{s}_0
$$

(where the free variables of the strictness expressions above belong to ($\vec{b}^-$,$\vec{b}_1^-$, $\vec{b}_2^-$).)

Clearly $C \cup C' \sim N \cup C'$. We shall now manipulate $N \cup C'$ (by means of correct transformations) with the aim of getting something of the desired form.

The first step is to exploit that $\vec{b}_2^- = \vec{b}_1^+$ and $\vec{b}_1^- = \vec{b}_2^+$, and afterwards exploit Fact 4.5 to replace the two inequalities for $\vec{b}_2^-$ by one. We arrive at

$$\begin{array}{lll} \vec{b}^+ \geq \vec{s} & \vec{b}_2^- \geq \vec{s}_1 \sqcup \vec{b}_5^- & \vec{b}_1^- \geq \vec{s}_2 \\ b_3^+ \geq \vec{s}_3 & \vec{b}_4^+ \geq \vec{s}_4 & \vec{b}_0 \gg \vec{s}_0 \\ \vec{b}_1^- = \vec{b}_2^+ & \vec{b}_2^- = \vec{b}_1^+ & \vec{b}_5^+ \geq \vec{b}_2^+ \end{array}$$

We now focus upon the pair of constraints

$$(\vec{b}_2^-, \vec{b}_1^-) \geq (\vec{s}_1 \sqcup \vec{b}_5^-, \vec{s}_2)$$

According to Fact 4.9, these can be replaced by the constraints

$$(\vec{b}_2^-, \vec{b}_1^-) \gg (\vec{S}_1, \vec{S}_2)$$

where $(\vec{S}_1, \vec{S}_2)$ is given as the "limit" of the chain with elements $(\vec{s}_{1n}, \vec{s}_{2n})$ given by

$$\begin{array}{rcl} (\vec{s}_{10}, \vec{s}_{20}) & = & \vec{0} \\ (\vec{s}_{1(n+1)}, \vec{s}_{2(n+1)}) & = & (\vec{s}_2 \sqcup \vec{b}_5^-, \vec{s}_2)[(\vec{s}_{1n}, \vec{s}_{2n})/(\vec{b}_2^-, \vec{b}_1^-)] \end{array}$$

This limit can be found as the k'th element, where $k = |(\vec{b}_1^-, \vec{b}_2^-)|$.

Our next step is to substitute in the new constraints for $\vec{b}_1^-$ and $\vec{b}_2^-$, using Fact 4.7 and Fact 4.8. We arrive at (after also having used Fact 4.6 to replace $=$ by $\gg$)

$$\begin{array}{lll} \vec{b}^+ \geq \vec{s}[(\vec{S}_1, \vec{S}_2)/(\vec{b}_2^-, \vec{b}_1^-)] & \vec{b}_2^- \gg \vec{S}_1 & \vec{b}_1^- \gg \vec{S}_2 \\ \vec{b}_3^+ \geq \vec{s}_3[(\vec{S}_1, \vec{S}_2)/(\vec{b}_2^-, \vec{b}_1^-)] & \vec{b}_4^+ \geq \vec{s}_4[(\vec{S}_1, \vec{S}_2)/(\vec{b}_2^-, \vec{b}_1^-)] & \vec{b}_0 \gg \vec{s}_0[(\vec{S}_1, \vec{S}_2)/(\vec{b}_2^-, \vec{b}_1^-)] \\ \vec{b}_2^+ \gg \vec{b}_1^- & \vec{b}_1^+ \gg \vec{b}_2^- & \vec{b}_5^+ \geq \vec{b}_2^+ \end{array}$$

Finally we use Fact 4.8 on the inequalities for $\vec{b}_2^+$ and $\vec{b}_1^+$, and subsequently use Fact 4.7 on the inequality for $\vec{b}_5^+$. At the same time we replace some $\geq$ by $\gg$, and arrive at

$$\begin{array}{lll}
\vec{b}^{+} \geq \vec{s}[(\vec{S}_1, \vec{S}_2)/(\vec{b}_2^{-}, \vec{b}_1^{-})] & \vec{b}_2^{-} \gg \vec{S}_1 & \vec{b}_1^{-} \gg \vec{S}_2 \\
\vec{b}_3^{+} \gg \vec{s}_3[(\vec{S}_1, \vec{S}_2)/(\vec{b}_2^{-}, \vec{b}_1^{-})] & \vec{b}_4^{+} \geq \vec{s}_4[(\vec{S}_1, \vec{S}_2)/(\vec{b}_2^{-}, \vec{b}_1^{-})] & \vec{b}_0 \gg \vec{s}_0[(\vec{S}_1, \vec{S}_2)/(\vec{b}_2^{-}, \vec{b}_1^{-})] \\
\vec{b}_2^{+} \gg \vec{S}_2 & \vec{b}_1^{+} \gg \vec{S}_1 & \vec{b}_5^{+} \geq \vec{S}_2
\end{array}$$

This is of the desired form, as it is quite easy to check that the only strictness variables occurring in the expressions on the right hand sides are those occurring in $\vec{b}^{-}$ and in $\vec{b}_5^{-}$.

4.4 Complexity Issues

One would of course like to estimate the complexity of the algorithm just developed. Before doing so, however, several issues have to be clarified, e.g.

- what should the input size parameter be? Natural choices could be the size of the expression, the size of its type, the maximal size of a subexpressions type, etc;
- how do we represent strictness expressions, and how do we perform the various manipulations on strictness expressions?

To give an in-depth treatment of these matters is beyond the scope of this paper, and hence we refrain from giving a complexity analysis...

5 Tanslating from CBN to CBV

In this section we give an example of the use of strictness annotations. We shall consider the following problem: given a λ-expression e annotated with strictness types, find a λ-expression e' such that e when evaluated using CBN yields the same result as e' when evaluated using CBV. The development of the translation algorithm will proceed in a number of steps.

The Mapping Z from $\mathcal{T}_{sa}$ into $\mathcal{T}$

If e can be assigned strictness type t, it (cf. Fact 3.3) has underlying type $\mathsf{E}(t)$. We cannot expect e', the translation of e into CBV, to have underlying type $\mathsf{E}(t)$ - rather e' should have type $Z(t)$, where Z besides removing annotations from arrows also thunkifies arguments to non-strict functions. That is, we have $Z(\mathsf{Base}) = \mathsf{Base}$, $Z(t_1 \to_0 t_2) = Z(t_1) \to Z(t_2)$, $Z(t_1 \to_1 t_2) = [Z(t_1)] \to Z(t_2)$.

The Mapping $C_t^{t'}$

To cope with the subsumption rule we need a function $C_t^{t'}$, parametrized by strictness types t and t' such that $t \leq t'$, with the following intended property: if e' has type $Z(t)$, then $C_t^{t'}(e')$ is an expression of type $Z(t')$.

Example 5.1 Suppose $t = \mathsf{Int} \to_0 \mathsf{Int}$ and $t' = \mathsf{Int} \to_1 \mathsf{Int}$, and suppose e' has type $\mathsf{Int} \to \mathsf{Int}$. $C_t^{t'}(e')$ then has to be something of type $[\mathsf{Int}] \to \mathsf{Int}$ – it is easily seen that $\lambda x.e'\ \mathcal{D}(x)$ (with x fresh) will do the job. $\square$

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

1. If $t = t'$, then $C_t^{t'}(e) = e$.
2. If $t = t_1 \to_0 t_2$ and $t' = t_1' \to_0 t_2'$ (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 $t = t_1 \to_1 t_2$ and $t' = t_1' \to_1 t_2'$ (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 $t = t_1 \to_0 t_2$ and $t' = t_1' \to_1 t_2'$ (with $t_1' \leq t_1, t_2 \leq t_2'$), then (where x is a "fresh" variable)
$$C_t^{t'}(e) = \lambda x.C_{t_2}^{t_2'}(e\ C_{t_1'}^{t_1}(\mathcal{D}(x)))$$

A Modified Inference System

It will be convenient to elaborate slightly on the inference system presented in Fig. 4. The resulting system is depicted in Fig. 8. The changes performed are:

- An extra entity T is introduced, such that judgements take the form $\Gamma', T \vdash_{sa} e \ : \ t, W$. Here T is a subset of the domain of Γ; the purpose of T is to record which variables have been bound by non-strict λ's.
- The two rules for abstraction and for application have been split into one for strict functions and one for non-strict.

The Translation Algorithm

Given an expression e, and a proof of $\Gamma, T \vdash_{sa} e \ : \ t, W$ using the rules in Fig. 8. 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 $\Gamma, T \vdash_{sa} e \ : \ t', W'$ because $\Gamma, T \vdash_{sa} e \ : \ t, W$ and $t \leq t', W \leq W'$. Suppose e (by the latter proof tree) transforms into e'. Then e (by the former proof tree) transforms into $C_t^{t'}(e')$.
- Suppose $e = c$. Then we let $e' = c$.
- Suppose $e = x$. 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:

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

$$\Gamma, T \vdash_{sa} c : CT_{sa}(c), \vec{1}$$

$$(\Gamma_1, (x : t), \Gamma_2), T \vdash_{sa} x : t, (\vec{1}, 0, \vec{1})$$

$$\frac{((x : t_1), \Gamma), T \vdash_{sa} e : t, (0, W)}{\Gamma, T \vdash_{sa} \lambda x.e : t_1 \rightarrow_0 t, W}$$

$$\frac{((x : t_1), \Gamma), T \cup \{x\} \vdash_{sa} e : t, (1, W)}{\Gamma, T \vdash_{sa} \lambda x.e : t_1 \rightarrow_1 t, W}$$

$$\frac{\Gamma, T \vdash_{sa} e_1 : t_2 \rightarrow_0 t_1, W_1 \ \Gamma, T \vdash_{sa} e_2 : t_2, W_2}{\Gamma, T \vdash_{sa} e_1 e_2 : t_1, W} \text{ if } W(x) = W_1(x) \sqcap W_2(x) \text{ for all } x$$

$$\frac{\Gamma, T \vdash_{sa} e_1 : t_2 \rightarrow_1 t_1, W_1 \ \Gamma, T \vdash_{sa} e_2 : t_2, W_2}{\Gamma, T \vdash_{sa} e_1 e_2 : t_1, W} \text{ if } W(x) = W_1(x) \text{ for all } x$$

$$\frac{\Gamma, T \vdash_{sa} e_1 : \mathsf{Bool}, W_1 \ \Gamma, T \vdash_{sa} e_2 : t, W_2 \ \Gamma, T \vdash_{sa} e_3 : t, W_3}{\Gamma, T \vdash_{sa} (\mathsf{if}\ e_1\ e_2\ e_3) : t, W}$$
$$\text{if } W(x) = W_1(x) \sqcap (W_2(x) \sqcup W_3(x)) \text{ for all } x$$

$$\frac{((f : t), \Gamma), T \vdash_{sa} e : t, (b, W)}{\Gamma, T \vdash_{sa} (\mathsf{rec}\ f\ e) : t, W}$$

Figure 8: An inference system for strictness types.

  - 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 $e = \mathsf{rec}\ f\ e_1$, and suppose e_1 (using the relevant proof tree) translates into e'_1. Then e translates into $\mathsf{rec}\ f\ e'_1$.

We see that if all arrows are annotated 1 (and correspondingly all λ-bound variables belong to T), we get the same code as produced by the "naive approach" (cf. Sect. 1.3). But if we are able to annotate some arrows 0, better code (i.e. fewer thunkifications/dethunkifications) will be obtained.

Example 5.2 Consider the expression ***twice*** $= \lambda f.\lambda x.f(fx)$. ***twice*** has strictness type

$$(\mathsf{Int}\rightarrow_1\mathsf{Int})\rightarrow_0(\mathsf{Int}\rightarrow_1\mathsf{Int}) \text{ because}$$
$$((f : \mathsf{Int}\rightarrow_1\mathsf{Int})), \emptyset \vdash_{sa} \lambda x.\ f(fx) : (\mathsf{Int}\rightarrow_1\mathsf{Int}), \{f\} \text{ and}$$
$$((x : \mathsf{Int}),(f := \mathsf{Int}\rightarrow_1\mathsf{Int})),\{x\} \vdash_{sa} f(fx) : \mathsf{Int}, \{f\} \text{ etc.}$$

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. □

6 Proving the Translation Correct

Before proving the translation correct, one of course has to define what correctness means – this will be the topic of the next section.

6.1 Correctness Predicates

Initially we shall consider closed expressions only. We will 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).
- $q \sim_{t_1 \to_0 t_2} q_1'$ holds iff for all $q_2 q_2'$ such that $q_2 \sim_{t_1} q_2'$ we have $q_1 q_2 \sim_{t_1} q_1' q_2'$.
- $q \sim_{t_1 \to_1 t_2} q_1'$ holds iff for all $q_2 q_2'$ such that $q_2 \sim_{t_1} q_2'$ we have $q_1 q_2 \sim_{t_1} q_1' \underline{q_2'}$.

This very much resembles a logical relation, but notice the difference between $\sim_{t_1 \to_0 t_2}$ and $\sim_{t_1 \to_1 t_2}$.

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 and where $\Gamma, T \vdash_{sa} e : t, W$. We shall need an auxiliary function Z_T, mapping from $\mathcal{T}_{sa}$-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 6.1 $e\ \mathsf{COR}(t, W, \Gamma, T)\ e'$ holds iff (with $\{x_1 \ldots x_n\}$ being the domain of Γ)

1. $Z_T(\Gamma) t \vdash e' : Z(t)$.
2. $\mathsf{FV}(e) = \mathsf{FV}(e')$.
3. Let in the following closed terms q_i, q_i' ($i \in \{1 \ldots n\}$) be such that $q_i \sim_{\Gamma(x_i)} q_i'$. Then it must hold that

   $$e[\{q_1 \ldots q_n\}/\{x_1 \ldots x_n\}] \sim_t e'[\{Q_1' \ldots Q_n'\}/\{x_1 \ldots x_n\}]$$

   where Q_i' is defined as follows: if $x_i \in T$ then $Q_i' = \underline{q_i'}$ else $Q_i' = q_i'$.

   Suppose now that for some i we have that $W(x_i) = 0$ and $q_i \sim_{\Gamma(i)} \Omega$. Then

$$e[\{q_1 \ldots q_n\}/\{x_1 \ldots x_n\}] \sim_t \Omega$$

The first part of 3 resembles the standard way of extending relations from closed terms to open terms; the second part of 3 expresses that the variables which by W are mapped into 0 in fact are "needed".

6.2 Introducing Bounded Recursion

For technical reasons we shall introduce *bounded recursion*. That is, we add constructs of form $\mathsf{rec}_n\ f\ e$ (with $n \geq 0$) to the language (the old construct $\mathsf{rec}\ f\ e$ is now termed *unbounded recursion*). This device is motivated by the SOS-rule $\mathsf{rec}\ f\ e \Rightarrow_N e[(\mathsf{rec}\ f\ e)/f]$, where we want to (inductively) use properties of the latter rec to prove properties of the former rec.

Wrt. typing properties, rec_n behaves exactly as rec. Wrt. translation, $\mathsf{rec}_n\ f\ e$ translates into $\mathsf{rec}_n\ f\ e'$ (where e' is the translation of e). Wrt. semantics, we have the SOS-rules

$$\begin{array}{rcl} \mathsf{rec}_0\ f\ e & \Rightarrow_N & \mathsf{rec}_0\ f\ e \\ \mathsf{rec}_{n+1}\ f\ e & \Rightarrow_N & e[\mathsf{rec}_0\ f\ e]/f] \end{array}$$

and similarly for $\Rightarrow_V^*$.

6.3 Correctness theorems

The main effort will be to prove the following theorem:

Theorem 6.2 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 $e\ \mathsf{COR}(t, W, \Gamma, T)\ e'$.

As the user certainly will like to use unbounded recursion, Theorem 6.2 may seem to be of limited use. However, it actually enables us to prove what we are really looking for:

Theorem 6.3 Suppose q is a closed expression (which may contain unbounded recursion) such that $(), \emptyset \vdash_{sa} q \ : \mathsf{Base}, ()$. Let q' be the translation of q, using the algorithm in Section 5. Now for all constants (of base type) $c, q \Rightarrow_N^* c$ iff $c, q' \Rightarrow_V^* c$.

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 6.2 tells us that $q_n \ \mathsf{COR}(\mathsf{Base}, (), (), \emptyset) \ q'_n$. 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. □

We now embark on proving Theorem 6.2. The proof proceeds roughly speaking as follows:

1. In Sect. 6.4, 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. In Sect. 6.5, 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, in Sect. 6.6 we are able to prove Theorem 6.2 by induction in the proof tree.

6.4 Some Lemmas concerning $\sim_t$

Lemma 6.4 Let $q \sim_t q'$, let q_1 be of type $\mathsf{E}(t)$ and let q'_1 be of type $Z(t)$.

1. If q' and q'_1 both loop by $\Rightarrow_V, q \sim_t q'_1$.
2. If q and q_1 both loop by $\Rightarrow_N, q_1 \sim_t q'$.

PROOF: We only consider 1 (2 is analogous). We use induction in t; first consider when t is a base type. Then we can infer that q loops by $\Rightarrow_N$, implying that $q \sim_t q'_1$.

Next assume $t = t_2 \to_0 t_1$ (The case $t = t_2 \to_1 t_1$ is analogous). Then we must show that if $q_2 \sim_{t_2} q_2'$ then $qq_2 \sim_{t_1} q_1'q_2'$ But we know that $qq_2 \sim_{t_1} q'q_2'$. As $q'q_2'$ and $q_1'q_2'$ both loop by $\Rightarrow_V$, the induction hypothesis applied to t_1 gives the claim. $\Box$

Lemma 6.5 Let q be of type $\mathsf{E}(t)$ and let q' be of type $Z(t)$. If q loops by $\Rightarrow_N$ and q' loops by $\Rightarrow_V$ we have $q \sim_t q'$.

PROOF: Induction in t. If t is a base type, it is obvious. Now assume $t = t_2 \to_0 t_1$ (the case $t = t_2 \to_1 t_1$ is analogous). Then we have to show that $qq_2 \sim_{t_1} q'q_2'$, whenever $q_2 \sim_{t_2} q_2'$. But qq_2 loops by $\Rightarrow_N$ and $q'q_2'$ loops by $\Rightarrow_V$, so this follows from the induction hypothesis applied to t_1. $\Box$

Lemma 6.6 Suppose $q \Rightarrow_N q_1$. Then $q \sim_t q'$ iff $q_1 \sim_t q'$. Suppose $q' \Rightarrow_V q_1'$. Then $q \sim_t q'$ iff $q_1 \sim_t q_1'$.

PROOF: We only show the first part – by induction in t: For base types, it is obvious as $\Rightarrow_N$ is deterministic. So assume that $t = t_2 \to_0 t_1$ (the case $t = t_2 \to_1 t_1$ is analogous). It is enough if we can show that if $q_2 \sim_{t_2} q_2'$ then $qq_2 \sim_{t_1} q'q_2'$ iff $q_1q_2 \sim_{t_1} q'q_2'$. But as $qq_2 \Rightarrow_N q_1q_2$, the induction hypothesis yields the desired result. $\Box$

Lemma 6.7 For all constants c, $c \sim_{CT_{sa(c)}} c$.

PROOF: Induction in $t = CT_{sa}(c)$. If it is a base type, it is trivial. Otherwise, our requirements to constants tell us that $t = \mathsf{Base} \to_0 t_2$. So assume that $q \sim_{\mathsf{Base}} q'$ then we must show that $cq \sim_{t_2} cq'$. Two possibilities:

- There exists a constant w such that $q \Rightarrow_N^* w$. Then also $q' \Rightarrow_V^* w$. By Lemma 6.6 it will be enough to show that $cw \sim_{t_2} cw$, and again by applying Lemma 6.6 we see that it is enough to show that $\mathsf{Applycon}(c, w) \sim_{t_2} \mathsf{Applycon}(c, w)$. If $\mathsf{Applycon}(c, w)$ yields a constant this follows from the induction hypothesis applied to t_2 – if $\mathsf{Applycon}(c, w) = c\ w$ then use Lemma 6.5.
- q loops by $\Rightarrow_N$. Then also q' loops by $\Rightarrow_V$. This implies that $c\ q$ loops by $\Rightarrow_N$ and that $c\ q'$ loops by $\Rightarrow_V$, so we can apply Lemma 65. $\Box$

Lemma 6.8 Suppose that it holds that $q \sim_t q'$ implies that $e[q/f] \sim_t e'[q'/f]$ (the latter being closed terms). Then for all $n, \mathsf{rec}_n\ f\ e \sim_t \mathsf{rec}_n\ f\ e'$.

PROOF: Induction in n. If $n = 0$, both sides loop so we can apply Lemma 6.5. In the induction case, it by Lemma 6.6 is enough to show that

$$e[\mathsf{rec}_{n-1}\ f\ e/f] \sim_t e'[\mathsf{rec}_{n-1}\ f\ e'/f]$$

But this follows from the assumptions and the induction hypothesis. □

Lemma 6.9 Suppose it for all q such that there exists q' with $q \sim_t q'$ holds that $e[q/f] \sim_t \Omega$ (where $\mathsf{FV}(e) \subseteq \{x\}$.) Then for all n, $\mathsf{rec}_n\ f\ e \sim_t \Omega$.

PROOF: Induction in n. If $n = 0$, $\mathsf{rec}_n\ f\ e$ loops and the claim follows from Lemma 6.5. If $n > 0$, it by Lemma 6.6 is enough to show that $e[\mathsf{rec}_{n-1}\ f\ e/f] \sim_t \Omega$. But this follows from the assumption of the lemma and the induction hypothesis. □

6.5 Correctness of $C_t^{t'}$

Lemma 6.10 For all strictness types t and t' with $t \leq t'$, $C_t^{t'}$ satisfies the following properties:

1. $\mathsf{FV}(e) = \mathsf{FV}(C_t^{t'}(e))$.
2. With symbols having appropriate types,
$$C_t^{t'}(e)[\{q_1 \ldots q_n\}/\{x_1 \ldots x_n\}] = C_t^{t'}(e[\{q_1 \ldots q_n\}/\{x_1 \ldots x_n\}])$$
3. If $\Gamma \vdash e\ :\ Z(t)$, then $\Gamma \vdash C_t^{t'}(e)\ :\ Z(t')$.
4. Suppose $q \sim_t \Omega$. Then also $q \sim_{t'} \Omega$. (here $C_t^{t'}$ is not involved, but this is needed for technical purposes).
5. Suppose $q \sim_t q'$. Then also $q \sim_{t'} C_t^{t'}(q')$.

PROOF: Induction in t (in the "number of arrows"). That property 1 and 2 hold is easy to see. Now consider property 3. Three cases:

- $t = t_1 \rightarrow_0 t_2,\ t' = t'_1 \rightarrow_0 t'_2$. We know that $\Gamma \vdash e\ :\ Z(t_1) \rightarrow Z(t_2)$ and must show that

$$\Gamma \vdash \lambda x.C^{t'_2}_{t_2}(e\ C^{t_1}_{t'_1}))\ :\ Z(t'_1) \rightarrow Z(t'_2)$$

  But this will easily follow from the induction hypothesis.

- $t = t_1 \rightarrow_1 t_2,\ t' = t'_1 \rightarrow_1 t'_2$. We know that $\Gamma \vdash e\ :\ [Z(t_1)] \rightarrow Z(t_2)$, and must show that

$$\Gamma \vdash \lambda x.C^{t'_2}_{t_2}(e\ \underline{C^{t_1}_{t'_1}(\mathcal{D}(x))})\ :\ [Z(t'_1)] \rightarrow Z(t'_2)$$

  Also this follows easily from the induction hypothesis.

- $t = t_1 \rightarrow_0 t_2,\ t' = t'_1 \rightarrow_1 t'_2$. We know that $\Gamma \vdash e\ :\ Z(t_1) \rightarrow Z(t_2)$ and must show that

$$\Gamma \vdash \lambda x.C^{t'_2}_{t_2}(e\ C^{t_1}_{t'_1}(\mathcal{D}(x)))\ :\ [Z(t'_1)] \rightarrow Z(t'_2)$$

  which follows easily from the induction hypothesis.

Next for property 4, where we can assume that $t = t_1 \rightarrow_0 t_2$ and $t' = t'_1 \rightarrow_1 t'_2$ with $t'_1 \leq t_1$, $t_2 \leq t'_2$ (the other cases are similar). We know that $q \sim_t \Omega$. In order to verify that $q \sim_{t'} \Omega$, assume that $q_1 \sim_{t'_1} q'_1$ – then we must show that $q\ q_1 \sim_{t'_2} \Omega\underline{q'_1}$. Inductively we can assume that property 5 holds for t'_1, so $q_1 \sim_{t_1} C^{t_1}_{t'_1}$. Then $q\ q_1 \sim_{t_2} \Omega C^{t_1}_{t'_1}(q'_1)$ i.e. (by Lemma 6.4) that $q\ q_1 \sim_{t_2} \Omega$. Inductively we can assume that property 4 holds for t_2, so $q\ q_1 \sim_{t'_2} \Omega$ – which by Lemma 6.4 is the desired result.

Finally, we have to check that property 5 holds. 3 cases:

- $t = t_1 \rightarrow_0 t_2,\ t' = t'_1 \rightarrow_1 t'_2$. Since $q \sim_t q'$ we know that

$$q_1 \sim_{t_1} q'_1 \text{ implies that } qq_1 \sim_{t_2} q'q'_1 \tag{1}$$

  Assuming that $q_1 \sim_{t'_1} q'_1$, our task is to show that

$$qq_1 \sim_{t'_2} (\lambda x C^{t'_2}_{t_2}(q'\ C^{t_1}_{t'_1}(x)))\ q'_1 \tag{2}$$

  Now two possibilities:

– q_1' loops by $\Rightarrow_V$. By Lemma 6.4, $q_1 \sim_{t_1'} \Omega$. Inductively we can assume that property 4 holds for t_1', so $q_1 \sim_{t_1} \Omega$. By (1), this implies that $qq_1 \sim_{t_2} q'\Omega$. By Lemma 6.4, this (as $q'\Omega$ loops by $\Rightarrow_V$) amounts to $qq_1 \sim_{t_2} \Omega$. But by Lemma 6.4, this is just (2).

– There exists a w_1' in WHNF such that $q_1' \Rightarrow_V^* w_1'$. By repeated application of Lemma 6.6, it in order to show (2) is enough to show that $qq_1 \sim_{t_2'} C_{t_2}^{t_2}(q'\ C_{t_1'}^{t_1'}(x))[w_1'/x]$ which (as we can assume that property 2 holds) amounts to showing that $qq_1 \sim_{t_2'} C_{t_2}^{t_2}(q'\ C_{t_1'}^{t_1'}(w_1'))$. By the induction hypothesis applied to t_2, this can be done by showing that $qq_1 \sim_{t_2'} q'\ C_{t_1'}^{t_1'}(w_1')$. As (1) holds, it is enough to show that $q_1 \sim_{t_1} C_{t_1'}^{t_1}(w_1')$ which by the induction hypothesis applied to t_1' can be done by showing $q_1 \sim_{t_1'} w_1'$ But this follows (from $q_1 \sim_{t_1'} q_1'$) by repeated application of Lemma 6.6.

- $t = t_1 \rightarrow_1 t_2$, $t' = t_1' \rightarrow_1 t_2'$. Since $q \sim_t q'$ we know that

$$q_1 \sim_{t_1} q_1' \text{ implies that } qq_1 \sim_{t_2} q'\underline{q_1'} \tag{3}$$

Assuming that $q_1 \sim_{t_1'} q_1'$, our task is to show that

$$qq_1 \sim_{t_2'} (\lambda x.C_{t_2}^{t_2'}(q'\ \underline{C_{t_1'}^{t_1}(\mathcal{D}(x))))}\ \underline{q_1'}.$$

Lemma 6.6 says (as $\underline{q_1'}$ is in WHNF) that is sufficient to show that $qq_1 \sim_{t_2'} C_{t_2}^{t_2'}(q'\ \underline{C_{t_1'}^{t_1}(\mathcal{D}(x))})[\underline{q_1'}/x]$ which (as we can assume that property 2 holds) amounts to showing that $qq_1 \sim_{t_2'} C_{t_2}^{t_2'}(q'\ \underline{C_{t_1'}^{t_1}(\mathcal{D}(\underline{q_1'}))})$. By the induction hypothesis applied to t_2, this can be done by showing that $qq_1 \sim_{t_2} q'\ \underline{C_{t_1'}^{t_1}(\mathcal{D}(\underline{q_1'}))}$. As (3) holds, it is enough to show that $q_1 \sim_{t_1} C_{t_1'}^{t_1}(\mathcal{D}(\underline{q_1'}))$ which by the induction hypothesis applied to t_1' can be done by showing $q_1 \sim_{t_1'} \mathcal{D}(\underline{q_1'})$. But this follows (from $q_1 \sim_{t_1'} q_1'$) by Lemma 6.6 and Fact 2.2.

- $t = t_1 \rightarrow_0 t_2$, $t' = t_1' \rightarrow_1 t_2'$. Since $q \sim_t q'$ we know that

$$q_1 \sim_{t_1} q_1' \text{ implies that } qq_1 \sim_{t_2} q'q_1' \tag{4}$$

Assuming that $q_1 \sim_{t_1'} q_1'$ our task is to show that

$$qq_1 \sim_{t'_2} (\lambda x C^{t'_2}_{t_2}(q'\ C^{t_1}_{t'_1}(\mathcal{D}(x))))\underline{q'_1}.$$

Lemma 6.6 says (as $\underline{q'_1}$ is in WHNF) that is sufficient to show that $qq_1 \sim_{t'_2} C^{t'_2}_{t_2}(q'\ C^{t_1}_{t'_1}(\mathcal{D}(x)))[\underline{q'_1}/x]$ which (as we can assume that property 2 holds) amounts to showing that $qq_1 \sim_{t'_2} C^{t'_2}_{t_2}(q'\ C^{t_1}_{t'_1}(\mathcal{D}(\underline{q'_1})))$. By the induction hypothesis applied to t_2, this can be done by showing that $qq_1 \sim_{t'_2}\ q'\ C^{t_1}_{t'_1}(\mathcal{D}(\underline{q'_1}))$. As (4) holds, it is enough to show that $q_1 \sim_{t_1} C^{t_1}_{t'_1}(\mathcal{D}(\underline{q'_1}))$ which by the induction hypothesis applied to t'_1 can be done by showing $q_1 \sim_{t'_1} \mathcal{D}(\underline{q'_1})$. But this follows (from $q_1 \sim_{t'_1} q'_1$) by Lemma 6.6 and Fact 2.2.

□

6.6 Proof of Theorem 6.2

We will proceed by induction in the proof tree for

$$\Gamma, T \vdash_{sa} e\ :\ t, W$$

(using the inference system in Fig. 8). The non-trivial parts will be to show that

- $Z_T(\Gamma) \vdash e'\ :\ Z(t)$.
- If $q_i \sim_{\Gamma(x_i)} q'_i$ then

  $$e[\{q_1 \ldots q_n\}/\{x_1 \ldots x_n\}] \sim_t e'[\{Q'_1 \ldots Q'_n\}/\{x_1 \ldots x_n\}]$$

  where $Q'_i = \underline{q'_i}$ if $x_i \in T$; $Q'_i = q'_i$ otherwise. Moreover, if $W(x_i) = 0$ and $q_i \sim_{\Gamma(x_i)} \Omega$ then

  $$e[\{q_1 \ldots q_n\}/\{x_1 \ldots x_n\}] \sim_t \Omega$$

We split into several cases:

- Suppose $\Gamma, T \vdash_{sa} e \ : \ t', W'$ because $\Gamma, T \vdash_{sa} e \ : \ t, \ W, \ t \leq t', \ W' \geq W$. We must show that

$$Z_T(\Gamma) \vdash C_t^{t'}(e') \ : \ Z(t')$$

but by Lemma 6.10 this follows from the induction hypothesis.

Next we must show that if $q_i \sim_{\Gamma(x_i)} q_i'$, then

$$e[\{q_1 \ldots q_n\}/\{x_1 \ldots x_n\}] \sim_{t'} C_t^{t'}(e')[\{Q_1' \ldots Q_n'\}/\{x_1 \ldots x_n\}]$$

But again this follows from the induction hypothesis and Lemma 6.10.

Finally we must show that if $W'(x_i) = 0$ and $q_i \sim_{\Gamma(x_i)} \Omega$ then $e[\{q_1 \ldots q_n\}/\{x_1 \ldots x_n\}] \sim_{t'} \Omega$. But this follows from the induction hypothesis and Lemma 6.10, as $W' \geq W$ so also $W(x_i) = 0$.

- Suppose $\Gamma, T \vdash_{sa} c \ : \ t, \vec{1}$ with $t = CT_{sa}(c)$. c translates into c. We must show that $Z_t(\Gamma) \vdash c \ : \ Z(t)$ but this is obvious as $t = Ct(c)[\vec{0}, ()]$ so $Z(t) = Ct(c)$.

Next we must show that $c \sim_t c$ but this is the content of Lemma 6.7

- Suppose $\Gamma, T \vdash_{sa} x \ : \ t, W$ with $\Gamma = (\Gamma_1, (x \ : \ t), \Gamma_2), \ W = (\vec{1}, 0, \vec{1})$ Let $i0$ be such that $x = x_{i0}$. Two cases:

  - $x \notin T$: then x translates into x. We must show that $Z_T(\Gamma) \vdash x \ : \ Z(t)$ but this is obvious as $x \notin T$.

    Next we must show that if $q_i \sim_{\Gamma(x_i)} q_i'$ then

    $$x[\{q_1 \ldots q_n\}/\{x_1 \ldots x_n\}] \sim_t x[\{Q_1' \ldots Q_n'\}/\{x_1 \ldots x_n\}]$$

    which amounts to showing that $q_{i0} \sim_t q_{i0}'$ which follows trivially from the assumptions.

    Finally, we must show that if $q_{i0} \sim_{\Gamma(x)} \Omega$ then also $x[\{q_1 \ldots q_n\}/\{x_1 \ldots x_n\}] \sim_t \Omega$ – but this is trivial.
  - $x \in T$: then x translates into $\mathcal{D}(x)$. We must show that $Z_T(\Gamma) \vdash \mathcal{D}(x) \ : \ Z(t)$ but this is obvious since $Z_T(\Gamma)(x) = [Z_T(\Gamma(x))]$.

    Next we must show that if $q_i \sim_{\Gamma(x_i)} q_i'$ then

    $$x[\{q_1 \ldots q_n\}/\{x_1 \ldots x_n\}] \sim_t \mathcal{D}(x)[\{Q_1' \ldots Q_n'\}/\{x_1 \ldots x_n\}]$$

which amounts to showing that $q_{i0} \sim_t \mathcal{D}(\underline{q'_{i0}})$ which follows from Lemma 6.6 and Fact 2.2.

Finally, we must show that if $q_i \sim_{\Gamma(x)} \Omega$ then also $x[\{q_1 \dots q_n\}/\{x_1 \dots x_n\}] \sim_t \Omega$ – but this is trivial.

- Suppose $\Gamma, T \vdash_{sa} \lambda x.e \ : \ t_1 \to_0 t, \ W$ because $((x \ : \ t_1), \Gamma), T \vdash_{sa} e \ : \ t, (0, W)$. Here $\lambda x.e$ translates into $\lambda x.e'$, where e translates into e'.

  Our first task is to show that $Z_T(\Gamma) \vdash \lambda x.e' \ : \ Z(t_1 \to_0 t)$. This can be done by showing that $((x : Z(t_1)), Z_T(\Gamma)) \vdash e' : Z(t)$ but as $((x : Z(t_1), Z_T(\Gamma)) = Z_T((x : t_1), \Gamma)$ this follows from $e \ \mathsf{COR}(t, \_, \_, \_) \ e'$.

  Now suppose $q_i \sim_{\Gamma(x_i)} q'_i$. We have to show that

  $$\lambda x.(e[\{q_1 \dots q_n\}/\{x_1 \dots x_n\}]) \sim_{t_1 \to_0 t} \lambda x.(e'[\{Q'_1 \dots Q'_n\}/\{x_1 \dots x_n\}])$$

  Assuming that $q \sim_{t_1} q'$, this amounts to showing

  $$\lambda x.(e[\{q_1 \dots q_n\}/\{x_1 \dots x_n\}])q \sim_t \lambda x.(e'[\{Q'_1 \dots Q'_n\}/\{x_1 \dots x_n\}])q'$$

  Now we split into two cases:

  - Suppose q' loops by CBV. Then $q \sim_t \Omega$. Since $e \ \mathsf{COR}(t, (0, W), \_, \_)$ e', we can conclude that

    $$e[\{q, q_1 \dots q_n\}/\{x, x_1 \dots x_n\}]) \sim_t \Omega$$

    By Lemma 6.6, we can conclude that

    $$\lambda x.(e[\{q_1 \dots q_n\}/\{x_1 \dots x_n\}])q \sim_t \Omega$$

    and by Lemma 6.4 this yields the desired.

  - Suppose $q' \Rightarrow_V^* w'$, with w' in WHNF. Then our task (by Lemma 6.6) amounts to showing that

    $$e[\{q, q_1 \dots q_n\}/\{x, x_1 \dots x_n\}]) \sim_t e'[\{w', Q'_1 \dots Q'_n\}/\{x, x_1 \dots x_n\}]$$

    But since (by Lemma 6.6) $q \sim_{t_1} w'$, this follows from $e \ \mathsf{COR}(t, \_, \_, T)$ e' (with $x \notin T$).

  Finally, we have to show that if $W(x_i) = 0, q_i \sim_{\Gamma(x_i)} \Omega$ then

  $$\lambda x.e[\{q_1 \dots q_n\}/\{x_1 \dots x_n\}] \sim_{t_1 \to_0 t} \Omega$$

So assuming that $q \sim_t q'$, we have to show that

$$\lambda x.e[\{q_1 \ldots q_n\}/\{x_1 \ldots x_n\}])q \sim_t \Omega q'$$

which by Lemma 6.6 and Lemma 6.4 amounts to showing that

$$e[\{q, q_1 \ldots q_n\}/\{x, x_1 \ldots x_n\}]) \sim_t \Omega$$

But this follows from e $\mathsf{COR}(t, (0, W), \_, \_)e'$.

- Suppose $\Gamma, T \vdash_{sa} \lambda x.e \ : \ t_1 \rightarrow_1 t, W$ because $((x \ : \ t_1), \Gamma), T \cup \{x\} \vdash_{sa} e \ : \ t, (1, W)$. Here $\lambda x.e$ translates into $\lambda x.e'$, where e translates into e'.

  Our first task is to show that $Z_T(\Gamma) \vdash \lambda x.e' \ : \ Z(t_1 \rightarrow_1 t)$. This can be done by showing that $((x \ : \ [Z(t_1)]), Z_T(\Gamma)) \vdash e' \ : \ Z(t)$ but as $((x \ : \ [Z(t_1)]), Z_T(\Gamma)) = Z_{T \cup \{x\}}((x \ : \ t_1), \Gamma)$ this follows from e $\mathsf{COR}(t, \_, \_, \_)$ e'.

  Now suppose $q_i \sim_{\Gamma(x_i)} q_i'$. We have to show that

$$\lambda x.(e[\{q_1 \ldots q_n\}/\{x_1 \ldots x_n\}]) \sim_{t_1 \rightarrow_0 t} \lambda x.(e'[\{Q_1' \ldots Q_n'\}/\{x_1 \ldots x_n\}])$$

  Assuming that $q \sim_{t_1} q'$, this amounts to showing

$$\lambda x.(e[\{q_1 \ldots q_n\}/\{x_1 \ldots x_n\}])q \sim_t \lambda x.(e'[\{Q_1' \ldots Q_n'\}/\{x_1 \ldots x_n\}])\underline{q'}$$

  which (by Lemma 6.6) amounts to showing that

$$e[\{q, q_1 \ldots q_n\}/\{x, x_1 \ldots x_n\}] \sim_t e'[\{\underline{q'}, Q_1' \ldots Q_n'\}/\{x, x_1 \ldots x_n\}])$$

  But this follows from e $\mathsf{COR}(t, \_, \_, T \cup \{x\})e'$.

  Finally, we have to show that if $W(x_i) = 0, q_i \sim_{\Gamma(x_i)} \Omega$ then

$$\lambda x.e[\{q_1 \ldots q_n\}/\{x_1 \ldots x_n\}]) \sim_{t_1 \rightarrow_0 1^t} \Omega$$

  So assuming that $q \sim_t q'$, we have to show that

$$\lambda x.e[\{q_1 \ldots q_n\}/\{x_1 \ldots x_n\}])q \sim_t \Omega \underline{q'}$$

which by Lemma 6.6 and Lemma 6.4 amounts to showing that

$$e[\{q, q_1 \dots q_n\}/\{x_1 \dots x_n\}]) \sim_t \Omega$$

But this follows from $e\ \mathsf{COR}(t, W, \_, \_)\ e'$.

- Suppose $\Gamma, T \vdash_{sa} e_1 e_2\ :\ t_1, W$ because $\Gamma, T \vdash_{sa} e_1\ :\ t_2 \to_0 t_1, W_1$ and $\Gamma, T \vdash_{sa} e_2\ :\ t_2, W_2$ with $W(x) = 0$ iff $W_1(x) = 0$ or $W_2 = 0$. Here $e_1 e_2$ translates into $e_1' e_2'$, where e_1 translates into e_1' and e_2 translates into e_2'.

  Our first task is to show that $Z_T(\Gamma) \vdash e_1' e_2'\ :\ Z(t_1)$. But as e_1' and e_2' are correct translations, we have $Z_T(\Gamma) \vdash e_1'\ :\ Z(t_2 \to_0 t_1)$ and $Z_T(\Gamma) \vdash e_2'\ :\ Z(t_2)$ enabling us to conclude the desired.

  Next we must show that if $q_i \sim_{\Gamma(x_i)} q_i'$ then

$$(e_1 e_2)[\{q_1 \dots q_n\}/\{x_1 \dots x_n\}]) \sim_{t_1} (e_1' e_2')[\{Q_1' \dots Q_n'\}/\{x_1 \dots x_n\}]$$

  But this follows from e_1 and e_2 being correct translations, since

$$\begin{array}{c} e_1[\{q_1 \dots q_n\}/\{x_1 \dots x_n\}]) \sim_{t_2 \to_0 t_1} e_1'[\{Q_1' \dots Q_n'\}/\{x_1 \dots x_n\}] \text{ and} \\ e_2[\{q_1 \dots q_n\}/\{x_1 \dots x_n\}]) \sim_{t_2} e_2'[\{Q_1' \dots Q_n'\}/\{x_1 \dots x_n\}] \end{array}$$

  Finally we must show that if $W(x_i) = 0, q_i \sim_{\Gamma(i)} \Omega$ then $e_1 e_2[\{q_1 \dots q_n\}/\{x_1 \dots x_n\}] \sim_{t_1} \Omega$. We distinguish between two cases:

  - if $W_1(x_i) = 0$, then (as $e_1\ \mathsf{COR}(\_, W_1, \_, \_) e_1'$) it holds that $e_1[\{q_1 \dots q_n\}/\{x_1 \dots x_n\}] \sim_{t_2 \to_0 t_1} \Omega$. As $e_2\ \mathsf{COR}(\_, \_, \_, \_)\ e_2'$, we can conclude

$$\begin{array}{r} e_1[\{q_1 \dots q_n\}/\{x_1 \dots x_n\}] e_2[\{q_1 \dots q_n\}/\{x_1 \dots x_n\}] \sim_{t_1} \\ \Omega e_2[\{q_1 \dots q_n\}/\{x_1 \dots x_n\}] \end{array}$$

    which (by Lemma 6.4) gives the desired.
  - if $W_2(x_i) = 0$, then (as $e_2\ \mathsf{COR}(\_, W_2, \_, \_) e_2'$) it holds that $e_2[\{q_1 \dots q_n\}/\{x_1 \dots x_n\}] \sim_{t_2} \Omega$ and hence (as $e_1\ \mathsf{COR}(\_, \_, \_, \_)\ e_1'$)

$$\begin{array}{r} e_1[\{q_1 \dots q_n\}/\{x_1 \dots x_n\}] e_2[\{q_1 \dots q_n\}/\{x_1 \dots x_n\}] \sim_{t_1} \\ e_1'[\{Q_1' \dots Q_n'\}/\{x_1 \dots x_n\}] \Omega \end{array}$$

As $e'_1[\{Q'_1 \ldots Q'_n\}/\{x_1 \ldots x_n\}]\Omega$ loops by $\Rightarrow_V$, Lemma 6.4 gives the desired result.

- Suppose $\Gamma,\ T \vdash_{sa} e_1 e_2\ :\ t_1, W_1$ because $\Gamma,\ T \vdash_{sa} e_1\ :\ t_2 \to_1 t_1,\ W_1$ and $\Gamma,\ T \vdash_{sa} e_2\ :\ t_2, W_2$. Here $e_1 e_2$ translates into $e'_1 \underline{e'_2}$, where e_1 translates into e'_1 and e_2 translates into e'_2. Inductively, we can assume that
  $e_1\ \mathsf{COR}(t_{2\to_1 t_1}, W_1, \Gamma, T) e'_1$ and $e_2\ \mathsf{COR}(t_2, W_2, \Gamma, T) e'_2$.

  Our first task is to show that $Z_T(\Gamma) \vdash e'_1 \underline{e'_2}\ :\ Z(t_1)$. But as e'_1 and e'_2 are correct translations, we have $Z_T(\Gamma) \vdash e'_1\ :\ Z(t_2 \to_0 t_1) = [Z(t_2)] \to Z(t_1)$ and $Z_T(\Gamma) \vdash e'_2\ :\ Z(t_2)$ enabling us to conclude the desired.

  Next we must show that if $q_i \sim_{\Gamma(x_i)} q_i$ then

  $$(e_1 e_2)[\{q_1 \ldots q_n\}/\{x_1 \ldots x_n\}] \sim_{t_1} (e'_1 \underline{e'_2})[\{Q'_1 \ldots Q'_n\}/\{x_1 \ldots x_n\}]$$

  But this follows easily from e'_1 and e'_2 being correct translations.

  Finally we must show that if $W_1(x_i) = 0, q_i \sim_{\Gamma(i)} \Omega$ then

  $$e_1 e_2[\{q_1 \ldots q_n\}/\{x_1 \ldots x_n\}] \sim_{t_1} \Omega$$

  As $e_1\ \mathsf{COR}(\_, W_1, \_, \_)\ e'_1$ we have $e_1[\{q_1 \ldots q_n\}/\{x_1 \ldots x_n\} \sim_{t_2 \to_1 t_1} \Omega$.
  As $e_2\ \mathsf{COR}(\_, \_, \_, \_)\ e'_2$, we can conclude

  $$\begin{array}{rl} e_1[\{q_1 \ldots q_n\}/\{x_1 \ldots x_n\}] e_2[\{q_1 \ldots q_n\}/\{x_1 \ldots x_n\}] & \sim_{t_1} \\ \Omega \underline{e'_2[\{Q'_1 \ldots Q'_n\}/\{x_1 \ldots x_n\}]} & \end{array}$$

  which (by Lemma 6.4) gives the desired.

- Suppose $\Gamma, T \vdash_{sa} \mathsf{if}\ e_1\ e_2\ e_3\ :\ t, W$ because $\Gamma, T \vdash_{sa} e_1\ \mathsf{Bool},\ W_1, \Gamma, T \vdash_{sa} e_2\ :\ t, W_2, \Gamma, T \vdash_{sa} e_3\ :\ t, W_3$ and $W(x) = W_1(x) \sqcap (W_2(x) \sqcup W_3(x))$. Now $\mathsf{if}\ e_1\ e_2\ e_3$ translates into $\mathsf{if}\ e'_1\ e'_2\ e'_3$, where e_i translates into e'_i.

  Our first task is to show that $Z_T(\Gamma) \vdash \mathsf{if}\ e'_1\ e'_2\ e'_3\ :\ Z(t)$. But this is immediate, since the correctness of the e'_i's tells us that

  $Z_T(\Gamma) \vdash e'_1\ :\ Z(\mathsf{Bool}) = \mathsf{Bool}$ and $Z_T(\Gamma) \vdash e'_2\ :\ Z(t)$ and $Z_T(\Gamma) \vdash e'_1\ :\ Z(t)$

  Next assume that $q_i \sim_{\Gamma(x_i)} q'_i$, then we must show that

$$\begin{array}{rll} & \mathsf{if}\ e_1[\{q_1 \ldots q_n\}/\{x_1 \ldots x_n\}] & e_2[\{q_1 \ldots q_n\}/\{x_1 \ldots x_n\}] \\ & & e_3[\{q_1 \ldots q_n\}/\{x_1 \ldots x_n\}] \\ \sim_t & \mathsf{if}\ e_1'[\{Q_1' \ldots Q_n'\}/\{x_1 \ldots x_n\}] & e_2'[\{Q_1' \ldots Q_n'\}/\{x_1 \ldots x_n\}] \\ & & e_3'[\{Q_1' \ldots Q_n'\}/\{x_1 \ldots x_n\}] \end{array}$$

We distinguish between three cases:

- $e_1[\{q_1 \ldots q_n\}/\{x_1 \ldots x_n\}]$ loops (by $\Rightarrow_N$). Then (as e_1' is a correct translation of e_1) also $e_1'[\{Q_1' \ldots Q_n'\}/\{x_1 \ldots x_n\}]$ loops (by $\Rightarrow_V$). Now apply Lemma 6.5.
- $e_1[\{q_1 \ldots q_n\}/\{x_1 \ldots x_n\}] \Rightarrow_N^* \mathsf{True}$. Then also $e_1'[\{Q_1' \ldots Q_n'\}/\{x_1 \ldots x_n\}] \Rightarrow_V^* \mathsf{True}$. By repeated application of Lemma 6.6, it is enough to show that

  $$e_2[\{q_1 \ldots q_n\}/\{x_1 \ldots x_n\}] \sim_t e_2'[\{Q_1' \ldots Q_n'\}/\{x_1 \ldots x_n\}]$$

  but this follows from e_2' being a correct translation.
- $e_1[\{q_1 \ldots q_n\}/\{x_1 \ldots x_n\}] \Rightarrow_N^* \mathsf{False}$. This is analogous to the previous case.

Finally, we have to check that if $W(x_i) = 0$ and $q_i \sim_{\Gamma(x_i)} \Omega$ then

$$\begin{array}{ll} \mathsf{if}\ e_1[\{q_1 \ldots q_n\}/\{x_1 \ldots x_n\}] & e_2[\{q_1 \ldots q_n\}/\{x_1 \ldots x_n\}] \\ & e_3[\{q_1 \ldots q_n\}/\{x_1 \ldots x_n\}] \sim_t \Omega \end{array}$$

We split between two cases:

- $W_1(x_i) = 0$. Then (as $e_1\ \mathsf{COR}(\mathsf{Bool}, W_1, \_, \_)\ e_1'$) it holds that $e_1[\{q_1 \ldots q_n\}/\{x_1 \ldots x_n\}] \sim_{\mathsf{Bool}} \Omega$ which amounts to saying that $e_1[\{q_1 \ldots q_n\}/\{x_1 \ldots x_n\}]$ loops – hence the claim.
- $W_2(x_i) = 0$ and $W_3(x_i) = 0$. Then (as $e_2\ \mathsf{COR}(t, W_2, \_, \_)\ e_2'$ and $e_3\ \mathsf{COR}(t, W_3, \_, \_)$)

  $e_2[\{q_1 \ldots q_n\}/\{x_1 \ldots x_n\}] \sim_t \Omega$ and $e_3[\{q_1 \ldots q_n\}/\{x_1 \ldots x_n\}] \sim_t \Omega$

  If $e_1[\{q_1 \ldots q_n\}/\{x_1 \ldots x_n\}] \Rightarrow_N^* \mathsf{True}$, the claim follows from the above and Lemma 6.6. Similarly in the False-case.

  If $e_1[\{q_1 \ldots q_n\}/\{x_1 \ldots x_n\}]$ loops, the claim is trivial.

- Suppose $\Gamma, T \vdash_{sa} \mathsf{rec}_n\ f\ e\ :\ t, W$ because $((f\ :\ t), \Gamma), T \vdash_{sa} e\ :\ t, (b, W)$. Now $\mathsf{rec}_n\ f\ e$ translates into $\mathsf{rec}_n\ f\ e'$, where e' is the translation of e.

  First we have to check that $Z_T(\Gamma) \vdash \mathsf{rec}_n\ f\ e'\ :\ Z(t)$ but this is immediate since the correctness of e' tells us that $((f\ :\ Z(t)), Z_T(\Gamma)) \vdash e'\ :\ Z(t)$.

  Next assume that $q_i \sim_{\Gamma(x_i)} q_i'$, then we have to show that

  $$\mathsf{rec}_n\ f\ e[\{q_1 \ldots q_n\}/\{x_1 \ldots x_n\}] \sim_t \mathsf{rec}_n\ f\ e'[\{Q_1' \ldots Q_n'\}/\{x_1 \ldots x_n\}]$$

  This follows from Lemma 6.8, provided we can show that

  $q \sim_t q'$ implies that
  $e[\{q, q_1 \ldots q_n\}/\{f, x_1 \ldots x_n\}] \sim_t e'[\{q', Q_1' \ldots Q_n'\}\ \{f, x_1 \ldots x_n\}]$

  But this follows from e' being a correct translation of e.

  Finally, we have to show that if $W(x_i) = 0$ and $q_i \sim_{\Gamma(x_i)} \Omega$ then

  $$\mathsf{rec}_n\ f\ e[\{q_1 \ldots q_n\}/\{x_1 \ldots x_n\}] \sim_t \Omega$$

  This follows from Lemma 6.9, provided we can show that if $q \sim_t q'$ then

  $$e[\{q, q_1 \ldots q_n\}/\{f, x_1 \ldots x_n\}] \sim_t \Omega$$

  But this follows from e' being a correct translation of e.

This concludes the proof of Theorem 6.2.

7 Concluding Remarks

We believe the main contributions of this paper to be:

- one more (the second, the first being [Wan93]) application has been given of the paradigm: an analysis and a transformation exploiting this analysis should be proved correct simultaneously;

- a (we think) novel approach to constraint solving, based on normalizing constraints while distinguishing between co/contravariant polarity, has been presented.

ln order to give a more precise analysis one may consider annotating the function arrows with the free variables needed by the function, cf. Example 3.6.

And to avoid the kind of superfluous dethunkification/thunkification we encountered in Example 5.2, one may consider keeping track of context – somewhat similar to what is done in [NN90].

References

[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.

A An Implementation

Below we list the commented source code of a system implementing the type inference algorithm from Sect. 4 and the translation algorithm from Sect. 5. At the end we show some examples of the use of the system, including the examples given in the main text.

The author is *not* to be kept responsible for possible errors in the system...

```
|| This file contains an implementation of
|| the inference algorithm and the translation algorithm
|| described in DAIMI PB-448 by Torben Amtoft:
|| "Strictness Types: an Inference Algorithm and an Application"

|| Main structure of the system:
|| 0) the user supplies an expression where bound variables
||    are annotated with their underlying types.
||    Such an expression is represented as
||    something of type "exptree"?
|| 1) by means of the function "annotate" this expression
||    is transformed into something of type "exptree_ta",
||    i.e. an expression where all subexpressions are
||    annotated with their types.
||    This function is quite trivial.
|| 2) by means of the function "infersolve" this expression
||    is transformed into something of type "result_sa",
||    i.e. an expression where all subexpressions are
||    annotated with open strictness types,
||    together with a list of normalized constraints
||    among the strictness variables.
||    This function implements the heart of the
||    type inference algorithm in Section 4.
|| 3) the user supplies the values of the strictness variables
||    occurring in contravariant position in the overall type.
||    Then the function "miniann" transforms the expression
||    from 2) into something of type "exptree\_sta",
||    i.e. an expression where all subexpressions are
||    annotated with strictness types (and the minimal such).
||    This function is rather trivial.
|| 4) by means of the function "translate" this expression
||    is transformed into an expression of type "exptree", such
||    that the CBV-semantics of this expression equals
||    the CBN-semantics of the expression from 0.
||    This function implements the translation algorithm
||    from Section 5.
|| 5) by means of the functions "cbneval" and "cbveval"
||    expressions of type "exptree" can be evaluated
```

```
||    by CBN resp. CBV.

|| ============ DATA STRUCTURES ======================

|| ---- TYPES ----

typ ::= Boolt |
        Intt |
        Unitt |
        Arrow typ typ

int2int = Arrow Intt Intt

|| ---- STRICTNESS TYPES ----

styp ::= Ints |
         Bools |
         Arrow0 styp styp |
         Arrow1 styp styp

|| ---- STRICTNESS VARIABLES ----

strictvar == num

|| ----  STRICTNESS EXPRESSIONS ----
|| We shall represent strictness expressions
|| as conjunctive normal form, that is as
|| a list without duplicates
||   (with implicit conjunction, i.e. glb)
|| of sorted lists
||   (with implicit disjunction, i.e. lub)
|| of strictness variables.
|| If [] occurs in the list,
|| the list is the singleton [[]].
|| Hence 0 has the unique representation [[]]
|| and 1 has the unique representation [].

strictexp == [[strictvar]]
```

```
one = []

zero = [[]]

|| ---- EXPRESSIONS ----
exptree ::=
  Con contype |
  Var [char] |
  Abs ([char],typ) exptree |
  App exptree exptree |
  If exptree exptree exptree |
  Ret ([char],typ) exptree

|| constants are assumed to have one of the following types

contype ::=
  Unitc |
  Intc num |
  Int1c (num -> num) |
  Int2c (num -> num -> num) |
  Boolc bool |
  Bool1c (num -> bool) |
  Bool2c (num -> num -> bool)

|| ---- EXPRESSIONS ANNOTATED WITH TYPE INFORMATION ----

typeinfo == ([[char]], [typ],typ)

exptree_ta ::=
  Cona Typeinfo contype |
  Vara typeinfo [char] |
  Absa typeinfo [char] exptree_ta |
  Appa typeinfo exptree_ta exptree_ta |
  Ifa typeinfo exptree_ta exptree_ta exptree_ta |
  Reca typeinfo [char] exptree_ta

|| typeinfo records the free variables of the expression,
```

```
|| their types and the type of the expression.

|| ----  EXPRESSIONS ANNOTATED WITH OPEN STRICTNESS TYPES
||       TOGETHER WITH CONSTRAINTS ----

exptree_sa ::=
  Cons stritypeinfo contype |
  Vars stritypeinfo [char] |
  Abss stritypeinfo [char] exptree_sa |
  Apps stritypeinfo exptree_sa exptree_sa |
  Ifs stritypeinfo exptree_sa exptree_sa exptree_sa |
  Recs stritypeinfo ([char],[strictvar],[strictvar]) exptree_sa

stritypeinfo == (typ, [strictvar], [strictvar], [strictvar])
|| the type of the expression,
|| the strictness variables corresponding to the
|| covariant/contravariant arrows of the type,
|| and the strictness variables corresponding to W.

result_sa == (exptree_sa,typ,[strictvar],[strictvar],[strictexp])
|| besides the tree also the type of the expression,
|| the strictness variables corresponding to the
|| contravariant arrows of this type,
|| the left sides of the constraints
|| and the right sides of these constraints.

|| ---- EXPRESSIONS ANNOTATED WITH STRICTNESS TYPES ----

exptree_sta ::=
  Consa styp contype |
  Varsa styp [char] |
  Abssa styp ([char],styp) exptree_sta |
  Appsa styp exptree_sta exptree_sta |
  Ifsa styp exptree_sta exptree_sta exptree_sta |
  Recsa styp ([char],styp) exptree_sta

|| ================== AUXILIARY FUNCTIONS ====================
```

```
|| ---- GENERAL FUNCTIONS ----

quicksort [] = []
quicksort (a:x) =
  (quicksort (filter (< a) x)) ++ [a] ++ (quicksort (filter (> a) x))

map3 g [] [] [] = []
map3 g (x:xs) (y:ys) (z:zs) =
  ((g x y z) : (map3 g xs ys zs))

takedrop n xs = (take n xs, drop n xs)}

|| ---- HANDLING UNIQUE VARIABLES ----

makenew k nf = (take k (iterate (+ 1) nf),nf + k)

uniquevar nf = ("x" ++ (show nf), (nf + 1))

|| ---- HANDLING STRICTNESS EXPRESSIONS

interexp == [[strictexp]]
|| An interexp is
|| a list (with implicit conjunction)
|| of lists (with implicit disjunction)
|| of lists (with implicit conjunction)
|| of lists (with implicit disjunction)
|| of strictness variables

normalize :: interexp -> strictexp

normalize = mkunique . mkflat

mkflat :: interexp -> strictexp
|| "flattens" an expression,
|| employing among others the distributive law

mkflat =
  cdd2cd . ccdd2cdd . cdcd2ccdd
```

```
  where
    cdcd2ccdd = map dcd2cdd
    dcd2cdd [] = [[]]
    dcd2cdd (cl : cs) =
      [(c:d) | c <- c1; d <- ds]
      where ds = dcd2cdd cs
    ccdd2cdd = concat
    cdd2cd = map concat

mkunique :: strictexp -> strictexp
|| ensures that the "uniqueness" requirements
|| to strictexp are indeed fulfilled.

mkunique =
  testzero . mk_cs_unique . mk_ds_unique
  cdd2cd . ccdd2cdd . cdcd2ccdd
  where
    mk_ds_unique = map quicksort
    mk_cs_unique = mkset
    testzero se = [[]], if (or (map (= []) se))
    testzero se = se, otherwise

sv2se :: strictvar -> strictexp

sv2se b = [[b]]

substv :: [strictvar] -> [strictvar] -> strictexp -> strictexp

substv news olds se =
  mkunique ((map (map (sbst news olds))) se)
  where
  sbst [] [] b = b
  sbst (bn : bns) (bo : bos) b = bn, if b = bo
  sbst (bn : bns) (bo : bos) b = sbst bns bos b, otherwise

substs :: [strictexp] -> [strictvar] -> strictexp -> interexp

substs news olds se =
```

```
  (map (map (sbst news olds))) se
  where
  sbst [] [] b = sv2se b
  sbst (bn : bns) (bo : bos) b = bn, if b = bo
  sbst (bn : bns) (bo : bos) b = sbst bns bos b, otherwise

|| ---- HANDLING TYPES

noofcov :: typ -> num

noofcov Boolt = 0
noofcov Intt = 0
noofcov (Arrow t1 t2) = 1 + (noofcov t2) + (noofctr t1)

noofctr :: typ -> num

noofctr Boolt = 0
noofctr Intt = 0
noofctr (Arrow t1 t2) = (noofctr t2) + (noofcov t1)

thunk :: typ -> typ

thunk t = Arrow Unitt t

mkthunk :: exptree -> num -> (exptree,num)

mkthunk ex nf =
  (Abs (newx,Unitt) ex,nf')
  where
  (newx,nf') = uniquevar nf

dethunk :: exptree -> exptree

dethunk ex = (App ex (Con (Unitc)))

erase :: styp -> typ

erase Ints = Intt
```

```
erase Bools = Boolt

erase (Arrow0 t1 t2) = (Arrow (erase t1) (erase t2))

erase (Arrow1 t1 t2) = (Arrow (erase t1) (erase t2))

zerase :: styp -> typ

zerase Ints = Intt

zerase Bools = Boolt

zerase (Arrow0 t1 t2) = (Arrow (zerase t1) (zerase t2))

zerase (Arrow1 t1 t2) = (Arrow (thunk (zerase t1)) (zerase t2))

annotype :: typ -> [strictexp] -> [strictexp] -> styp
|| in annotate t sp sm, sp and sm are assumed to be zero or one.

annotype Intt sp sm = Ints
annotype Boolt sp sm = Bools
annotype (Arrow t1 t2) sp sm =
  t'
  where
  tp1 = noofcov t1
  tm1 = noofctr t1
  (sp1,(sparrow:sp2)) = takedrop tm1 sp
  (sm1,sm2) = takedrop tp1 sm
  t1' = annotype t1 sm1 sp1
  t2' = annotype t2 sp2 sm2
  t' = Arrow0 t1' t2', if sparrow = zero
  t' = Arrow1 t1' t2, if sparrow = one

|| ==================== PHASE 1: ANNOTATE ====================

annotate :: exptree -> exptree_ta
```

```
annotate ex =
  e'
  where
  (e',t) = annot ex [] []

annot (Con c) fvs ts =
  ((Cona (fvs,ts,t) c),t)
  where
  t = ctype c
  ctype (Unite) = Unitt
  ctype (Intc n) = Intt
  ctype (Int1c f) = Arrow Intt Intt
  ctype (Int2c f) = Arrow Intt (Arrow Intt Intt)
  ctype (Boolc b) = Boolt
  ctype (Bool1c f) = Arrow Intt Boo1t
  ctype (Bool2c f) = Arrow Intt (Arrow Intt Boolt)

annot (Var x) fvs ts =
  ((Vara (fvs,ts,t) x),t)
  where
  t = ts!k
  k = #(takewhile ((~) . (= x)) fvs)

annot (Abs (x,t) e1) fvs ts =
  ((Absa (fvs,ts,t') x e1'),t')
  where
  (e1',t1) = annot e1 (x:fvs) (t:ts)
  t' = Arrow t t1

annot (App e1 e2) fvs ts =
  ((Appa (fvs,ts,t) e1' e2'),t),
     if (isarrow t1) & ((source t1) = t2)
  where
  (e1',t1) = annot e1 fvs ts
  (e2',t2) = annot e2 fvs ts
  t = target t1
  isarrow (Arrow ta tb) = True
  isarrow ta = False, otherwise
```

```
  source (Arrow ta tb) = ta
  target (Arrow ta tb) = tb

annot (App e1 e2) fvs ts =
  error "type mismatch in application", otherwise

annot (If e1 e2 e3) fvs ts =
  ((Ifa (fvs,ts,t2) e1' e2' e3'),t2), if (t1 = Boolt) & (t2 = t3)
  where
  (e1',t1) = annot e1 fvs ts
  (e2',t2) = annot e2 fvs ts
  (e3',t3) = annot e3 fvs ts

annot (If e1 e2 e3) fvs ts =
  error "type mismatch in conditional", otherwise

annot (Rec (f,t) e1) fvs ts =
  ((Reca (fvs,ts,t) f e1'),t), if t1 = t
  where
  (e1',t1) = annot e1 (f:fvs) (t:ts)

annot (Ret (f,t) e1) fvs ts =
  error "type mismatch in recursion", otherwise

|| ===================== PHASE 2: INFERSOLVE ====================

infersolve :: exptree_ta -> result_sa

infersolve e1 =
  (e1',t,b1m,cl,cr)
  where
  (e1',t,b1p,b1m,b2p,nf',cl,cr) = infsol e1 [] [] 2

infsol :: exptree_ta -> [strictvar] -> [strictvar] -> num ->
          (exptree_sa,typ,[strictvar],[strictvar],[strictvar],
          num, [strictvar], [strictexp])

|| suppose Gamma[bm,bp] |- e: t[b1p,b1m], b2p.
```

```
|| Then infsol e bp bm nf = (e',t,b1p,b1m,b2p,nf',cl,cr)
|| where nf records the next unused strictness variable,
|| where cl is a list of strictness variables
||               starting with bp + b1p + b2p,
|| where cr is the corresponding list of strictness expressions
||      (so for the first ">=" is implicit
||      and for the remaining ">>" is implicit).

infsol (Cona (fvs,ts,t) c) bp bm nf =
  ((Cons (t,b1p,[],b2p) c), t,b1p, [],b2p,nf'',cl,cr)
  where
  tp = noofcov t
  (b1pnf') = makenew tp nf
  (b2p,nf'') = makenew (#fvs) nf'
  cl = bp ++ b1p ++ b2p
  cr = (rep (#bp) zero) ++
       (rep (#b1p) zero) ++
       (rep (#b2p) one)

infsol (Vara (fvs,ts,t) x) bp bm nf =
  ((Vars (t,b4p,b4m,b567p) x),t,b4p,b4m,b567p,nf''',cl,cr)
  where
  fv1 = takewhile ((~) . (= x)) fvs
  t1p = sum (map noofcov (take (#fv1) ts))
  t1m = sum (map noofctr (take (#fv1) ts))
  t2P = noofcov (ts!(#fv1))
  t2m = noofctr (ts!(#fv1))
  b1p = take t1m bp
  (b2p,b3p) = takedrop t2m (drop t1m bp)
  b2m = take t2p (drop t1p bm)
  (b4pnf') = makenew t2p nf
  (b4m,nf'') = makenew t2m nf'
  (b567p,nf''') = makenew (#fvs) nf''
  (b5p,(b6p:b7p)) = takedrop (#fv1) b567p
  cl = b1p ++ b2p ++ b3p ++ b4p ++ b5p ++ [b6p] ++ b7p
  cr = (rep (#b1p) zero) ++
       (map sv2se b4m) ++
       (rep (#b3p) zero) ++
```

```
        (map sv2se b2m) ++
        (rep (#b5p) one) ++
        [zero] ++
        (rep (#b7p) one)

infsol (Absa (fvs,ts,(Arrow t1 t)) x e1) bp bm nf =
  ((Abss ((Arrow t1 t),b1p ++ [b3p] ++ b2p,b1m ++ b2m,b4p) x e1'),
   (Arrow t1 t),b1p ++ [b3p] ++ b2p,b1m ++ b2m,b4p,nf1,cl,cr)
  where
  t1p = noofcov t1
  t1m = noofctr t1
  (b1p,nf') = makenew t1m nf
  (b1m,nf'') = makenew t1p nf'
  (e1',t,b2p,b2m,(b3p:b4p),nf1,cl1,cr1) =
     infsol e1 (b1p ++ bp) (b1m ++ bm) nf''
  b0 = drop (#b1p + (#bp) + (#b2p) + 1 + (#b4p)) cl1
  (s1,cr1') = takedrop (#b1p) cr1
  (s,cr1'') = takedrop (#bp) cr1'
  (s2,cr1''') = takedrop (#b2p) cr1''
  ((s3:s4),s0) = takedrop (1 + (#b4p)) cr1'''
  cl = bp ++ b1p ++ [b3p] ++ b2p ++ b4p ++ b0
  cr = s ++ s1 ++ [s3] ++ s2 ++ s4 ++ s0

infsol (Appa (fvs,ts,t1) e1 e2) bp bm nf =
  ((Apps (t1,b4p,b4m,b8p) e1' e2'),t1,b4p,b4m,b8p,nf',cl,cr)
  where
  (e1',(Arrow t2a t1),b234p,b24m,b5p,nf1,cl1,cr1) =
    infsol e1 bp bm nf
  (e2',t2,b6p,b6m,b7p,nf2,cl2,cr2) = infsol e2 bp bm nf1
  t2p = noofcov t2
  t2m = noofctr t2
  (b2p, (b3p : b4p)) = takedrop t2m b234p
  (b2m,b4m) = takedrop t2p b24m
  b0 = drop (#bp + (#b234p) + (#fvs)) cl1
  (sa,cr1') = takedrop (#bp) cr1
  (s2,cr1'') = takedrop (#b2p) cr1'
  (s3:s4,cr1''') = takedrop ((#b4p) + 1) cr1''
  (s5,s0) = takedrop (#fvs) cr1'''
```

```
  b1 = drop (#bp + (#b6p) + (#fvs)) cl2
  (sb,cr2') = takedrop (#bp) cr2
  (s6,cr2'') = takedrop (#b6p) cr2'
  (s7,s1) = takedrop (#fvs) cr2''
  (b8p,nf') = makenew (#fvs) nf2
  cl = bp ++ b4p ++ b8p ++ b2p ++ b2m ++ [b3p] ++ b5p ++
       b6p ++ b6m ++ b7p ++ b0 ++ bl
  cr = (map2 g sa sb) ++
       (map (normalize . (substs s'6 b2m)) s4) ++
       (map2 g8 s5 s7) ++
       s'2 ++
       s'6 ++
      [normalize (substs s'6 b2m s3)] ++
      (map (normalize . (substs s'6 b2m)) s5) ++
      s'6 ++
      s'2 ++
      (map (normalize . (substs s'2 b6m)) s7) ++
      (map (normalize . (substs s'6 b2m)) s0) ++
      (map (normalize . (substs s'2 b6m)) s1)
g sa_ sb_ = normalize [[normalize (substs s'6 b2m sa_),
                        normalize (substs s'2 b6m sb_)]]
g8 s5_ s7_ = normalize [[normalize (substs s'6 b2m s5_)],
                          [normalize (substs s'6 b2m s3),
                           normalize (substs s'2 b6m s7_)]]
(s'2,s'6) =
  takedrop (#b6m) ((iterate h (rep (t2p + t2m) zero))!(t2p + t2m))
h s2ns6n =
  map (normalize . (substs s2ns6n (b6m ++ b2m))) (s2 ++ s6)

infsol (Ifa (fvs,ts,t) e1 e2 e3) bp bm nf =
  ((Ifs (t,b8p,b8m,b9p) e1' e2' e3'),t,b8p,b8m,b9p,nf''',cl,cr)
  where
  (e1',Boolt,[], [],b3p,nf1,cl1,cr1) = infsol e1 bp bm nf
  (e2',t,b4p,b4m,b5p,nf2,cl2,cr2) = infsol e2 bp bm nf1
  (e3',tb,b6p,b6m,b7p,nf3,cl3,cr3) = infsol e3 bp bm nf2
  b0 = drop (#bp + (#fvs)) cl1
  (sa,cr1') = takedrop (#bp) cr1
  (s3,s0) = takedrop (#fvs) cr1'
```

```
  b1 = drop (#bp + (#b4p) + (#fvs)) cl2
  (sb,cr2') = takedrop (#bp) cr2
  (s4,cr2'') = takedrop (#b4p) cr2'
  (s5,s1) = takedrop (#fvs) cr2''
  b2 = drop (#bp + (#b6p) + (#fvs)) cl3
  (sc,cr3') = takedrop (#bp) cr3
  (s6,cr3'') = takedrop (#b6p) cr3'
  (s7,s2) = takedrop (#fvs) cr3''
  (b8p,nf') = makenew (#b6p) nf3
  (b8m,nf'') = makenew (#b6m) nf'
  (b9p,nf''') = makenew (#b7p) nf''
  cl = bp ++ b8p ++ b9p ++ b3p ++ b4p ++ b4m ++ b5p ++
       b6p ++ b6m ++ b7p ++ b0 ++ b1 ++ b2
  cr = (map3 g sa sb sc) ++
       (map2 g8 s4 s6) ++
       (map3 g9 s3 s5 s7) ++
       s3 ++
       (map (substv b8m b4m) s4) ++
       (map sv2se b8m) ++
       (map (substv b8m b4m) s5) ++
       (map (substv b8m b6m) s6) ++
       (map sv2se b8m) ++
       (map (substv b8m b6m) s7) ++
       s0 ++
       (map (substv b8m b4m) s1) ++
       (map (substv b8m b6m) s2)
g sa_ sb_ sc_ =
  normalize [[sa_,substv b8m b4m sb_,substv b8m b6m sc_]]
g8 s4_ s6_ =
  normalize [[substv b8m b4m s4_,substv b8m b6m s6_]]
g9 s3_ s5_ s7_ =
  normalize [[s3_],[substv b8m b4m s5_,substv b8m b6m s7_]]

infsol (Reca (fvs,ts,t) f e1) bp bm nf =
  ((Recs (t,b5p,b5m,b4p) (f,b1p,b1m) e1'),t,b5p,b5m,b4p,nf1'',cl,cr)
  where
  tp = noofcov t
  tm = noofctr t
```

```
  (b1m,nf') = makenew tp nf
  (b1p,nf'') = makenew tm nf'
  (e1',t,b2p,b2m,(b3p:b4p),nf1,cl1,cr1) =
      infsol e1 (b1p ++ bp) (b1m ++ bm) nf''
  b0 = drop (tm + (#bp) + tp + (#fvs + 1)) cl1
  (s1,cr1') = takedrop tm cr1
  (s,cr1'') = takedrop (#bp) cr1'
  (s2,cr1''') = takedrop tp cr1''
  ((s3:s4),s0) = takedrop (#fvs + 1) cr1'''
  (b5p,nf1') = makenew tp nf1
  (b5m,nf1'') = makenew tm nf1'
  cl = bp ++ b5p ++ b4p ++ b1p ++ b1m ++ b2p ++ b2m ++ [b3p] ++ b0
  cr = (map (normalize . (substs (s'1 ++ s'2) (b2m ++ b1m))) s) ++
       s'2 ++
       (map (normalize . (substs (s'1 ++ s'2) (b2m ++ b1m))) s4) ++
       s'1 ++
       s'2 ++
       s'2 ++
       s'1 ++
       [(normalize (substs (s'1 ++ s'2) (b2m ++ b1m) s3))] ++
       (map (normalize . (substs (s'1 ++ s'2) (b2m ++ b1m))) s0)
  (s'1,s'2) =
    takedrop (#b2m) ((iterate h (rep (tp + tm) zero))! (tp + tm))
  h s1ns2n = map (normalize .
                  (substs s1ns2n (b2m ++ b1m)))
                 ((map2 lubb5m s1 b5m) ++ s2)
             where lubb5m s1_ b5m_ = normalize [[s1_,sv2se b5m_]]

|| ==================== PHASE 3 : MINIANN ====================

miniann :: result_sa -> [strictexp] -> exptree_sta

miniann (e1,t,bm,cl,cr) vm =
  minan e1 lkup
  where
  cl' = bm ++ cl
  cr' = vm ++ (map (normalize . (substs vm bm)) cr)
  lkup b = g cl' cr' b
```

```
          where g (b1:bs) (c1:cs) b = c1, if b = b1
                g (b1:bs) (c1:cs) b = g bs cs b, otherwise

minan :: exptree_sa -> (strictvar -> strictexp) -> exptree_sta

minan (Cons (t,bp,[],bw) c) lkup =
  Consa (annotype t sp []) c
  where
  sp = map lkup bp

minan (Vars (t,bp,bm,bw) x) lkup =
  Varsa (annotype t sp sm) x
  where
  sp = map lkup bp
  sm = map lkup bm

minan (Abss (t,bp,bm,bw) x e1) lkup =
  Abssa (annotype t sp sm) (x,annotype t1 sm1 sp1) e1'
  where
  e1' = minan e1 lkup
  sp = map lkup bp
  sm = map lkup bm
  (Arrow t1 t2) = t
  tp1 = noofcov t1
  tm1 = noofctr t1
  sp1 = take tm1 sp
  sm1 = take tp1 sm

minan (Apps (t,bp,bm,bw) e1 e2) lkup =
  Appsa (annotype t sp sm) e1' e2'
  where
  sp = map lkup bp
  sm = map lkup bm
  e1' = minan e1 lkup
  e2' = minan e2 lkup

minan (Ifs (t,bp,bm,bw) e1 e2 e3) lkup =
  Ifsa (annotype t sp sm) e1' e2' e3'
```

```
  where
  sp = map lkup bp
  sm = map lkup bm
  e1' = minan e1 lkup
  e2' = minan e2 lkup
  e3' = minan e3 lkup

minan (Recs (t,bp,bm,bw) (f,bp1,bm1) e1) lkup =
  Recsa (annotype t sp sm) (f,annotype t sm1 sp1) e1'
  where
  sp = map lkup bp
  sm = map lkup bm
  sp1 = map lkup bp1
  sm1 = map lkup bm1
  e1' = minan e1 lkup

|| ===================== PHASE 4 : TRANSLATE ====================

cvex :: styp -> styp -> exptree -> num -> (exptree,num)

cvex t t' ex nf = (ex,nf), if t = t'

cvex (Arrow0 t1 t2) (Arrow0 t1' t2') ex nf =
  (Abs (newx,(zerase t1')) ex', nf2)
  where
  (newx,nf') = uniquevar nf
  (ct1x,nf1) = cvex t1' t1 (Var newx) nf'
  (ex',nf2) = cvex t2 t2' (App ex ct1x) nf1

cvex (Arrow1 t1 t2) (Arrow1 t1' t2') ex nf =
  (Abs (newx,thunk (zerase t1')) ex', nf2)
  where
  (newx,nf') = uniquevar nf
  (ct1x,nf1) = cvex t1' t1 (dethunk (Var newx)) nf'
  (ct1x' ,nf1') = mkthunk ct1x nf1
  (ex',nf2) = cvex t2 t2' (App ex ct1x') nf1'

cvex (Arrow0 t1 t2) (Arrow1 t1' t2') ex nf =
```

```
  (Abs (newx,thunk (zerase t1')) ex', nf2)
  where
  (newx,nf') = uniquevar nf
  (ct1x,nf1) = cvex t1' t1 (dethunk (Var newx)) nf'
  (ex',nf2) = cvex t2 t2' (App ex ct1x) nf1

translate :: exptree sta -> exptree
|| implements the translation from section 5,
|| with the modification that
|| the subsumption rule is inlined in all rules
|| except the abstraction rule and the application rule.
|| That is, we use the inference system from Fig. 5.

translate e1 =
  e1'
  where
  (e1',t,nf) = trans e1 [] [] [] 2

trans :: exptree_sta -> [[char]] -> [styp] -> [[char]] -> num
                     -> (exptree,styp,num)
|| Suppose ((x1:t1)...(xn:tn)),T |- e: t, W and
|| suppose e translates into e'.
|| Then trans e (x1..xn) (t1..tn) T nf = (e',t,nf').

trans (Consa t c) fvs ts tv nf =
  (e',t,nf')
  where
  (e',nf') = cvex t' t (Con c) nf
  te = erase t
  tp = noofcov te
  t' = annotype te (rep tp zero) []

trans (Varsa t x) fvs ts tv nf =
  (e',t,nf')
  where
  (e',nf') = cvex t' t varx nf
  fv1 = takewhile ((~) . (= x)) fvs
  t' = ts!(#fv1)
```

```
  varx = dethunk (Var x), if member tv x
  varx = (Var x), otherwise

trans (Abssa (Arrow0 t1a t) (x,t1) e1) fvs ts tv nf =
  ((Abs (x,zerase t1) e1'),(Arrow0 t1a t),nf')
  where
  (e1',ta,nf') = trans e1 (x:fvs) (t1:ts) tv nf

trans (Abssa (Arrow1 t1a t) (x,t1) e1) fvs ts tv nf =
  ((Abs (x,thunk (zerase t1)) e1'),(Arrow1 t1a t),nf')
  where
  (e1',ta,nf') = trans e1 (x:fvs) (t1:ts) (x:tv) nf

trans (Appsa t e1 e2) fvs ts tv nf =
  (App e1' e2''',t,nf2')
  where
  (e1',t1,nf1) = trans e1 fvs ts tv nf
  (e2',t2,nf2) = trans e2 fvs ts tv nf1
  (e2'',nf2') = mkthunk e2' nf2
  e2''' = e2', if iszero t1
  e2''' = e2'', otherwise
  iszero (Arrow0 ta tb) = True
  iszero (Arrow1 ta tb) = False

trans (Ifsa t e1 e2 e3) fvs ts tv nf =
  (If e1' e2'' e3'',t,nf3')
  where
  (e1',Bools,nf1) = trans e1 fvs ts tv nf
  (e2',t2,nf2) = trans e2 fvs ts tv nf1
  (e2'',nf2') = cvex t2 t e2' nf2
  (e3',t3,nf3) = trans e3 fvs ts tv nf2'
  (e3'',nf3') = cvex t3 t e3', nf3

trans (Recsa t' (f,t) e1) fvs ts tv nf =
  (Rec (f,zerase t) e1'',t',nf')
  where
  (e1',ta,nf1) = trans e1 (f:fvs) (t:ts) tv nf
  (e1'',nf') = cvex t t' e1' nf1
```

```
|| ===================== PHASE 5: CBNEVAL and CBVEVAL ==================

whnf : exptree -> bool

whnf (Con c) = True

whnf Abs (x,t) e1) = True

whnf e1 = False, otherwise

subst :: exptree -> exptree -> [char] -> exptree

subst (Var x) e' y = e', if x = y
                   = (var x), otherwise

subst (Con c) e' y = (Con c)

subst (Abs (x,t) e1) e', y
  = (Abs (x,t) e1), if x = y
  = (Abs (x,t) (subst e1 e' y)), otherwise

subst (App e1 e2) e' y =
  (App e1, e2')
  where
  e1' = subst e1 e' y
  e2' = subst e2 e' y

subst (If e1 e2 e3) e' y =
  (If e1' e2' e3')
  where
  e1' = subst e1 e' y
  e2' = subst e2 e' y
  e3' = subst e3 e' y

subst (Rec (f,t) e1) e' y
  = (Rec (f,t) e1), if f = y
  = (Rec (f,t) (subst e1 e' y)), otherwise
```

```
consteval (Int2c f) (Intc n) = (Int1c (f n))

consteval (Int1c f) (Intc n) = (Intc (f n))

consteval (Bool2c f) (Intc n) = (Bool1c (f n))

consteval (Bool1c f) (Intc n) = (Boolc (f n))

cbneval :: exptree -> exptree

cbneval e1 = e1, if whnf e1

cbneval e1 = cbneval (cbn e1), otherwise

cbn (App (Abs (x,t) e1) e2) = subst e1 e2 x

cbn (App (Con c1) (Con c2)) = (Con (consteval c1 c2))

cbn (App (Con c1) e2) = (App (Con c1) (cbn e2))

cbn (App e1 e2) = (App (cbn e1) e2)

cbn (If (Con (Boolc True)) e2 e3) = e2

cbn (If (Con (Boolc False)) e2 e3) = e3

cbn (If e1 e2 e3) = (If (cbn e1) e2 e3), otherwise

cbn (Rec (f,t) e1) = subst e1 (Rec (f,t) e1) f

cbveval :: exptree -> exptree

cbveval e1 = e1, if whnf e1

cbveval e1 = cbveval (cbv e1), otherwise

cbv (App (Abs (x,t) e1) e2) = subst e1 e2 x, if whnf e2
```

```
cbv (App (Con c1) (Con c2)) = (Con (consteval c1 c2))

cbv (App e1 e2) = (App e1 (cbv e2)), if whnf e1

cbv (App e1 e2) = (App (cbv e1) e2), otherwise

cbv (If (Con (Boolc True)) e2 e3) = e2

cbv (If (Con (Boolc False)) e2 e3) = e3

cbv (If e1 e2 e3) = (If (cbv e1) e2 e3) , otherwise

cbv (Rec (f,t) e1) = subst e1 (Rec (f,t) e1) f

|| ==================== OVERALL SYSTEM ====================

cbn2cbv :: exptree -> [strictexp] -> exptree

cbn2cbv e1 bm = translate (miniann (infersolve (annotate e1)) bm)

|| ==================== TEST SETS WITH COMMENTS ===========

|| test = ((Lam x. Lam y. x) 7) omega

omega = Rec ("f" ,Intt) (Var "f")

test = App (App (Abs ("x",Intt)
                     (Abs ("y",Intt)
                          (Var "x")))
                (Con (Intc (7) )))
           omega

testt = cbn2cbv test []

|| Value of testt:
||
|| App (App (Abs ("x",Intt)
```

```
||          (Abs ("y",Arrow Unitt Intt)
||               (Var "x" ) ) )
||          (Con (Intc 7)))
||     (Abs ("x3",Unitt)
||          (Rec ("f" ,Intt) (Var "f")))
||
|| we see that "y" is thunkified but "x" is not.

testcbn = cbneval test

|| Value of testcbv:
||   Con (Intc 7)

testcbv = cbveval testt

|| Value of testcbv:
||   Con (Intc 7)

||   twice = Lam f. Lam x. f( f( x))

twice = Abs ("f",int2int)
            (Abs ("x",Intt)
                 (App (Var "f")
                      (App (Var "f")
                           (Var "x"))))

twices = infersolve (annotate twice)

|| Value of twices:
||
|| (Abss (Arrow (Arrow Intt Intt) (Arrow Intt Intt),[14,13],[2],[])
||       "f"
||       (Abss (Arrow Intt Intt,[13],[],[14])
||             "x"
||             (Apps (Intt, [], [],[13,14])
||                   (Vars (Arrow Intt Intt,[3],[], [4,5]) "f")
||                   (Apps (Intt, [], [], [11,12])
||                         (Vars (Arrow Intt Intt,[6], [], [7,8]) "f")
```

```
||                          (Vars Untt, [], [], [9,10]) "x"))),
|| Arrow (Arrow Intt Intt) (Arrow Intt Intt),
|| [2],
|| [14,13,3,4,5,11,12,6,7,8,9,10],
|| [[[]],[[2]],[[2]],[],[[]],[[2]],[[]],[[2]],[],[[]],[[]],[]])
||
|| Compare with Example 4.3.
|| Here the strictness variable numbered 2 plays the role of b1-,
|| the strictness variable numbered 14 plays the role of b11+
|| and the strictness variable numbered 13 plays the role of b12+.
|| It was predicted that there would be a constraint b12+ >= b1-,
|| that is a constraint "13 >= 2". And so there is, as can be
|| seen from the fact that "13" occurs second in the list of
|| left sides and "[[2]]" occurs second in the list of right sides.

twice0 = miniann twices [zero]

|| Value of twice0:
||           (the first line shows that given a strict function,
||            twice returns a strict function)
||   Abssa (Arrow0 (Arrow0 Ints Ints) (Arrow0 Ints Ints))
||         ("f",Arrow0 Ints Ints)
||         (Abssa (Arrow0 Ints Ints)
||                ("x", Ints)
||                (Appsa Ints
||                       (Varsa (Arrow0 Ints Ints) "f")
||                       (Appsa Ints
||                              (Varsa (Arrow0 Ints Ints) "f")
||                              (Varsa Ints "x"))))

twice1 = miniann twices [one]

|| Value of twice1:
||           (the first line shows that given a non-strict function,
||            twice returns a non-strict function)
|| Abssa (Arrow0 (Arrow1 Ints Ints) (Arrow1 Ints Ints))
||       ("f",Arrow1 Ints Ints)
||       (Abssa (Arrow1 Ints Ints)
```

```
||              ("x", Ints)
||              (Appsa Ints
||                     (Varsa (Arrow1 Ints Ints) "f")
||                     (Appsa Ints
||                            (Varsa (Arrow1 Ints Ints) "f")
||                            (Varsa Ints "x"))))

twice0t = translate twice0

|| Value of twice0t:
||
||   Abs ("f" ,Arrow Intt Intt)
||       (Abs ("x",Intt)
||            (App (Var "f")
||                 (App (Var "f")
||                      (Var "x"))))
||
|| Compare with Example 5.2.
|| As predicted, twice translates into itself.

twice1t = translate twice1

|| Value of twice1t:
||
||   Abs ("f" ,Arrow (Arrow Unitt Intt) Intt)
||     (Abs ("x",Arrow Unitt Intt)
||          (App (Var "f")
||               (Abs ("x3",Unitt)
||                    (App (Var "f")
||                         (Abs ("x2",Unitt)
||                              (App (Var "x")
||                                   (Con Unitc)))))))
||
|| Compare with Example 5.3 -- the translation is as predicted.


|| twiceid = twice (Lam x. x)
```

```
twiceid = App twice
              (Abs ("x",Intt)
                   (Var "x"))

twiceids = infersolve (annotate twiceid)

|| Value of twiceids:
||
|| (Apps
||    (Arrow Intt Intt,[13],[],[])
||    (Abss
||       (Arrow (Arrow Intt Intt) (Arrow Intt Intt),
||               [14,13,[2],[])
||       "f"
||       (Abss (Arrow Intt Intt, [13],[],[14])
||          "x"
||          (Apps (Intt, [],[],[13,14])
||             (Vars (Arrow Intt Intt,[3],[],[4,5]) "f")
||             (Apps (Intt, [], [],[11,12])
||                (Vars (Arrow Intt Intt,[6],[],[7,8]) "f")
||                (Vars (Arrow Intt Intt,[],[],[9,10]) "x")
||
||    (Abss
||       (Arrow Intt Intt,[15],[],[])
||       "x"
||       (Vars Untt, [],[],[15]) "x")),
|| Arrow Intt Intt,
|| [],
|| [13,2,14,15,3,4,5,11,12,6,7,8,9,10],
|| [[[]],[[]],[[]],[[]],[[]],[],[[]],
||  [[]],[[]],[[]],[],[[]],[[]],[]])
||
|| Compare with Example 4.4.
|| Here the strictness variable numbered 2 plays the role of b01-,
|| the strictness variable numbered 14 plays the role of b01+,
|| the strictness variable numbered 13 plays the role of b1+ and
|| the strictness variable numbered 15 plays the role of b02+.
|| It was predicted that the constraints could be normalized to
```

```
|| include "b1+ >= 0, b02+ >> 0", that is "13 >= 0" and "15 >> 0".
|| And so there is, as can be seen from the fact that "13" ("15")
|| occurs first (fourth) in the list of left sides and
|| "[[]]" occurs first (fourth) in the list of right sides.


twiceidst = miniann twiceids []

|| Value of twiceidst:
||   Appsa (Arrow0 Ints Ints)
||         (Abssa (Arrow0 (Arrow0 Ints Ints) (Arrow0 Ints Ints))
||                ("f",Arrow0 Ints Ints)
||                (Abssa (Arrow0 Ints Ints)
||                       ("x" Ints)
||                       (Appsa Ints
||                              (Varsa (Arrow0 Ints Ints) "f")
||                              (Appsa Ints
||                                     (Varsa (Arrow0 Ints Ints) "f")
||                                     (Varsa Ints "x")))))
||         (Abssa (Arrow0 Ints Ints)
||                ("x", Ints)
||                (Varsa Ints "x"))

twiceidt = translate twiceidst

|| Value of twiceidt:
||   App (Abs ("f" ,Arrow Intt Intt)
||            (Abs ("x",Intt)
||                 (App (Var "f")
||                      (App (Var "f")
||                           (Var "x")))))
||      (Abs ("x",Intt)
||           (Var "x"))

|| Example 3.5.

ex3_5 = Rec ("f", (Arrow Intt (Arrow Intt (Arrow Intt Intt))))
            (Abs ("x",Intt)
```

```
                (Abs ("y", Intt)
                     (Abs ("z",Intt)
                          (If (App (App (Con (Bool2c (=)))
                                        (Var "z"))
                                   (Con (Intc (0))))
                              (App (App (Con (Int2c (+)))
                                        (Var "x"))
                                   (Var "y"))
                              (App (App (App (Var "f")
                                             (Var "y"))
                                        (Var "x"))
                                   (App (App (Con (Int2c (-)))
                                             (Var "z"))
                                        (Con (Intc (1)))))))))

ex3_5st  = miniann (infersolve (annotate ex3_5)) []

|| Value of ex3_5st:
||
||          (the first line shows that our system,
||          as predicted, is able to deduce that
||          the function is strict in all arguments)
||
||   Recsa (Arrow0 Ints (Arrow0 Ints (Arrow0 Ints Ints)))
||     ("f",Arrow0 Ints (Arrow0 Ints (Arrow0 Ints Ints)))
||     (Abssa (Arrow0 Ints (Arrow0 Ints (Arrow0 Ints Ints)))
||       ("x",Ints)
||       (Abssa (Arrow0 Ints (Arrow0 Ints Ints))
||          ("y",Ints)
||          (Abssa (Arrow0 Ints Ints)
||             ("z",Ints)
||             (Ifsa Ints
||                (Appsa Bools
||                   (Appsa (Arrow0 Ints Bools)
||                      (Consa (Arrow0 Ints (Arrow0 Ints Bools))
||                         (Bool2c <function>))
||                      (Varsa Ints "z"))
||                   (Consa Ints (Intc 0)))
```

```
||                 (Appsa Ints
||                    (Appsa (Arrow0 Ints Ins)
||                       (Consa (Arrow0 Ints (Arrow0 Ints Ints))
||                          (Int2c <function>))
||                       (Varsa Ints "x"))
||                    (Varsa Ints "y"))
||                 (Appsa Ints
||                    (Appsa (Arrow0 Ints Ints)
||                       (Appsa (Arrow0 Ints (Arrow0 Ints Ints))
||                          (Varsa
||                             (Arrow0 Ints
||                                     (Arrow0 Ints
||                                             (Arrow0 Ints Ints)))
||                             "f")
||                          (Varsa Ints "y"))
||                       (Varsa Ints "x"))
||                    (Appsa Ints
||                       (Appsa (Arrow0 Ints Ints)
||                          (Consa (Arrow0 Ints (Arrow0 Ints Ints))
||                             (Int2c <function>))
||                          (Varsa Ints "z"))
||                       (Consa Ints (Intc 1))))))))

|| Example 3.6.

ex3_6 = Abs ("y",Intt)
            (Rec ("f",Arrow Intt Intt)
                 (Abs ("x",Intt)
                      (If (App (App (Con (Bool2c (=)))
                                    (Var "x"))
                               (Con (Intc (0))))
                          (Var "y")
                          (App (Var "f")
                          (App (App (Con (Int2c (-)))
                                    (Var "x"))
                               (Con (Intc (1)))))))))

ex3_6st = miniann (infersolve (annotate ex3_6)) []
```

```
|| Value of ex3_6st:
||
||            (the first line shows that our system,
||             as predicted, is unable to deduce that
||             the function is strict in its first argument)
||
|| Abssa (Arrow1 Ints (Arrow0 Ints Ints))
||   ("y",Ints)
||   (Recsa (Arrow0 Ints Ints)
||      ("f",Arrow0 Ints Ints)
||      (Abssa (Arrow0 Ints Ints)
||         ("x" ,Ints)
||         (Ifsa Ints
||            (Appsa Bools
||               (Appsa (Arrow0 Ints Bools)
||                  (Consa (Arrow0 Ints (Arrow0 Ints Bools))
||                     (Bool2c <function>))
||                  (Varsa Ints "x"))
||               (Consa Ints (Intc 0)))
||            (Varsa Ints "y")
||            (Appsa Ints
||               (Varsa (Arrow0 Ints Ints) "f")
||               (Appsa Ints
||                  (Appsa (Arrow0 Ints Ints)
||                     (Consa
||                        (Arrow0 Ints
||                                (Arrow0 Ints Ints))
||                        (Int2c <function>))
||                     (Varsa Ints "x"))
||                  (Consa Ints (Intc 1)))))))
```