Strictness and Totality Analysis

Kirsten Lackner Solberg*
Hanne Riis Nielson and Flemming Nielson
Computer Science Dept.
Aarhus University, Denmark
e-mail: {kls,hrn,fn}@daimi.aau.dk

Abstract

We define a novel inference system for strictness and totality analysis for the simply-typed lazy lambda-calculus with constants and fixpoints. Strictness information identifies those terms that definitely denote bottom (i.e. do not evaluate to WHNF) whereas totality information identifies those terms that definitely do not denote bottom (i.e. do evaluate to WHNF). The analysis is presented as an annotated type system allowing conjunctions only at "top-level". We give examples of its use and prove the correctness with respect to a natural-style operational semantics.

1 Introduction

Strictness analysis has proved useful in the implementation of lazy functional languages as Miranda, Lazy ML and Haskell: when a function is strict it is safe to evaluate its argument before performing the function call. Totality analysis is equally useful but has not be adopted so widely: if the argument to a function is known to terminate then it is safe to evaluate it before performing the function call [11].

In the literature there are several approaches to the specification of strictness analysis: abstract interpretation (e.g. [12, 4]), projection analysis (e.g. [22]) and inference based methods (e.g. [2, 8, 9, 10, 23]). Totality analysis has received much less attention and has primarily been specified using abstract interpretation [12, 1]. It can be regarded as an approximation to time complexity analysis; most literature performing such developments consider eager languages but [15] considers lazy languages.

In this paper we present an inference system for performing strictness *and* totality analysis. We restrict our attention to a simply typed lambda-calculus with constants and a fixpoint operator. The inference system is an extension of the usual type system in that we introduce three annotations on types t:

*Dept. of Math. and Computer Science, Odense University, Denmark

- $!^{\mathbf{b}}$t: the value has type t and it definitely $\perp$,
- $!^{\mathbf{n}}$t: the value has type t and is definitely *not* $\perp$, and
- $!^{\top}$t: the value has type t and it can be any value.

Annotated types can be constructed using the function type constructor and (top-level) conjunction. As an example a function may have the annotated type $(!^{\mathbf{n}}\mathtt{Int} \rightarrow !^{\mathbf{n}}\mathtt{Int}) \wedge (!^{\mathbf{b}}\mathtt{Int} \rightarrow !^{\mathbf{b}}\mathtt{Int})$ which means that given a terminating argument the function will definitely terminate and given a non-terminating argument it will definitely not terminate. Thus we capture the strictness as well as the totality of the function. Strictness and totality information can also be combined as in $(!^{\mathbf{b}}\mathtt{Int} \rightarrow !^{\mathbf{n}}\mathtt{Int} \rightarrow !^{\mathbf{n}}\mathtt{Int}) \wedge (!^{\mathbf{n}}\mathtt{Int} \rightarrow !^{\mathbf{b}}\mathtt{Int} \rightarrow !^{\mathbf{n}}\mathtt{Int}) \wedge (!^{\mathbf{b}}\mathtt{Int} \rightarrow !^{\mathbf{b}}\mathtt{Int} \rightarrow !^{\mathbf{b}}\mathtt{Int})$ which will be the annotated type of McCarthy's ambiguity operator.

The inference based approach allows to combine the two analyses. Mycroft [12] presents both analyses using abstract interpretation but the semantic foundations are different: the strictness analysis is based on downward closed sets and the totality analysis on upward closed sets. We believe that the two analyses could be combined using the convex power-domains of [13] but this will be untractable for two reasons. One is that the mathematical foundations will be rather complicated and extensions to richer languages would not be easy. Another reason is that implementations based on abstract interpretation often are rather inefficient due to the local computation of fixpoints and we would like to explore the use of other approaches that seem to offer better performance.

The semantic foundations of our work is based on natural style operational semantics. We employ a lazy semantics so terms are evaluated to weak head normal form (WHNF). This means we capture the semantics of "real-life" lazy functional languages in contrast to most other papers on strictness analysis like [4] where terms are evaluated to head normal forms. Since we are based on operational semantics fixpoint induction is not available for free and in the soundness proof for the analysis we shall use the trick of annotating the fixpoint operators with the number of unfoldings allowed.

In general, we follow the current trend of separating the specification of the analysis from its algorithmic realisation. The specification is often by means of an annotated types system and comes in one of two flavours. In the effect systems only type constructors are annotated and examples of analyses specified in this vain are [17, 18, 19, 23]. Our analysis belongs to the other group where subcomponents of types are annotated; further analyses in this group are [2, 8, 9, 10]. Inference based methods have also been used for variations of strictness and totality analysis; examples include [5] that uses a type system with intersection types to determine "neededness" of redexes and [3] that studies liveness properties.

Overview Section 2 presents the natural-style operational semantics and the standard type inference rules for our simply-typed lazy lambda calculus. Based on these (so-called underlying) types we construct (in Section 3) the strictness and totality types and give rules for coercing between them; also a notion of conjunction type is defined but

$$[\mathrm{var}]_{UT}\ \frac{}{A \vdash_{UT} x : ut} \text{ if } x : ut \in A$$

$$[\mathrm{abs}]_{UT}\ \frac{A, x : ut_1 \vdash_{UT} e : ut_2}{A \vdash_{UT} \lambda x.e : ut_1 \to ut_2}$$

$$[\mathrm{app}]_{UT}\ \frac{A \vdash_{UT} e_1 : ut_1 \to ut_2 \quad A \vdash_{UT} e_2 : ut_1}{A \vdash_{UT} e_1\ e_2 : ut_2}$$

$$[\mathrm{cond}]_{UT}\ \frac{A \vdash_{UT} e_1 : \mathtt{Bool} \quad A \vdash_{UT} e_2 : ut \quad A \vdash_{UT} e_3 : ut}{A \vdash_{UT} \mathrm{cond}\ e_1\ e_2\ e_3 : ut}$$

$$[\mathrm{fix}]_{UT}\ \frac{A \vdash_{UT} e : ut \to ut}{A \vdash_{UT} \mathrm{fix}\ e : ut}$$

$$[\mathrm{const}]_{UT}\ \frac{}{A \vdash_{UT} c : ut_c}$$

Figure 1: Type inference

only at "top-level"; finally the inference system is presented and examples of its use are given. In Section 4 we then present the correctness proof.

2 Syntax and Semantics

This section introduces the simply-typed lazy λ-calculus with constants and fixpoints. The underlying types, $ut \in UT$, are either base types or function types

$$ut ::= A \mid ut \to ut$$

and the base types (the A's) include `Bool` and `Int`. The terms, $e \in E$, of the simply-typed λ-calculus are

$$e ::= x \mid \lambda x.e \mid e\ e \mid \mathrm{fix}\ e \mid \mathrm{cond}\ e\ e\ e \mid c$$

where the constants (the c's) include true and false of type `Bool` and all integers of type `Int`. We only consider terms that are typeable according to the type inference rules defined in Figure 1 and we shall require that the bound variables in terms are all different. The list A of assumptions gives underlying types to free variables and for each constant c there is an underlying type ut_c. The set of free variables in the term e is written FV(e) and the usual substitution on terms is written $e[e_2/x]$.

The semantics will be lazy except that all built-in functions will be strict in each argument. Figure 2 defines a natural-style operational semantics. Terms are evaluated

$$[\text{app1}]\ \frac{\vdash e_1 \Downarrow \lambda x.e \quad \vdash e[e_2/x] \Downarrow v}{\vdash e_1\ e_2 \Downarrow v}$$

$$[\text{app2}]\ \frac{\vdash e_1 \Downarrow c \quad \vdash e_2 \Downarrow w}{\vdash e_1\ e_2 \Downarrow u}\ \text{if } (w, u) \in \text{meaning}(c)$$

$$[\text{fix}]\ \frac{\vdash e\ (\text{fix } e) \Downarrow v}{\vdash \text{fix } e \Downarrow v}$$

$$[\text{abs}]\ \frac{}{\vdash \lambda x.e \Downarrow \lambda x.e} \qquad [\text{const}]\ \frac{}{\vdash c \Downarrow c}$$

$$[\text{condT}]\ \frac{\vdash e_1 \Downarrow \text{true} \quad \vdash e_2 \Downarrow v_2}{\vdash \text{cond}\ e_1\ e_2\ e_3 \Downarrow v_2} \qquad [\text{condF}]\ \frac{\vdash e_1 \Downarrow \text{false} \quad \vdash e_3 \Downarrow v_3}{\vdash \text{cond}\ e_1\ e_2\ e_3 \Downarrow v_3}$$

Figure 2: Lazy semantics for closed terms

to WHNF, i.e. to constants or lambda-abstractions. The meaning of a constant c is given by a set meaning(c) of pairs of constants and the idea is that if $(u, v) \in$ meaning(c) then c u = v; e.g. $(2, +_2) \in$ meaning(+) and $(1, 3) \in$ meaning($+_2$). As mentioned in the introduction the semantics is faithful to current lazy languages like Miranda [20] and this is unlike other approaches (e.g. [4]) where terms are evaluated to HNF rather than WHNF. As usual we shall regard α-equivalent terms to be equal.

Two closed terms are semantically equivalent, written $e_1 \sim_{ut} e_2$, if they both evaluate to the same WHNF and have the same underlying type:

Definition 1 $(e_1 \sim_{ut} e_2) \Leftrightarrow ((\vdash e_1 \Downarrow w) \Leftrightarrow (\vdash e_2 \Downarrow w))$
provided both $\emptyset \vdash_{UT} e_1 : ut$ and $\emptyset \vdash_{UT} e_2 : ut$ can be inferred. □

We shall assume throughout the paper that there are no empty types, i.e. for each underlying type there exists a *terminating* term with that type. Clearly, for each type there exists a non-terminating term of that type, for example fix (λx.x).

3 Totality Types and Conjunction Types

We will now define the strictness and totality analysis for the simply-typed lazy λ-calculus. First we introduce the totality types and the coercions between them. On top of this we define the conjunction types. Finally we give the inference system for the combined strictness and totality analysis.

Totality types

A (strictness and) totality type, $t \in T$, is either an annotated underlying type or a function type between totality types:

$$t ::= !^{s}\mathrm{ut} \mid t \to t$$

The underlying type $\varepsilon(t)$ of a totality type t is obtained by erasing all annotations. The annotations (the s's) can either be $\top$, $\mathbf{n}$, or $\mathbf{b}$. The idea is that a term with the totality type $!^{\mathbf{b}}\mathrm{ut}$ has the underlying type ut and *does not* evaluate to a WHNF. A term with the totality type $!^{\mathbf{n}}\mathrm{ut}$ has the underlying type ut and *does* evaluate to a WHNF. Finally a term with the totality type $!^{\top}\mathrm{ut}$ has the underlying type ut but we do not know anything about the evaluation of the term. A term with the totality type $t_1 \to t_2$ will, when applied to a term with totality type t_1, yield a term with totality type t_2. We do not allow strictness and totality types of the form $!^{s}(t_1 \to t_2)$ where t_1 and t_2 are totality types since such a type is equivalent to the type $!^{s}(\varepsilon(t_1) \to \varepsilon(t_2)) \wedge (t_1 \to t_2)$ and we wish to deal separately with the complication of conjunction. (In this paper it will be allowed at the "top-level" only.)

Example 2 All functions with the underlying type $\mathrm{ut}_1 \to \mathrm{ut}_2$ will also have the totality types $!^{\top}(\mathrm{ut}_1 \to \mathrm{ut}_2)$ and $!^{\top}\mathrm{ut}_1 \to !^{\top}\mathrm{ut}_2$. A function with no WHNF has the totality type $!^{\mathbf{b}}(\mathrm{ut}_1 \to \mathrm{ut}_2)$ and the function that applied to any term yields a term with no WHNF has the totality type $!^{\top}\mathrm{ut}_1 \to !^{\mathbf{b}}\mathrm{ut}_2$. □

Later we shall need the predicate $\mathrm{BOT_T}(t)$ defined by

$$\begin{aligned}
\mathrm{BOT_T}(!^{\mathbf{b}}\mathrm{ut}) &= \mathtt{tt} & \qquad \mathrm{BOT_T}(!^{\top}\mathrm{ut}) &= \mathtt{tt} \\
\mathrm{BOT_T}(!^{\mathbf{n}}\mathrm{ut}) &= \mathtt{ff} & \qquad \mathrm{BOT_T}(t_1 \to t_2) &= \mathrm{BOT_T}(t_2)
\end{aligned}$$

The idea is that it holds whenever the totality type must incorporate a term without WHNF.

Coercions between totality types

Most terms have more than one totality type; as an example the totality types of $\lambda x.7$ include $!^{\top}(\mathtt{Int} \to \mathtt{Int})$, $!^{\mathbf{n}}(\mathtt{Int} \to \mathtt{Int})$, and $!^{\top}\mathtt{Int} \to !^{\mathbf{n}}\mathtt{Int}$. Some of these are redundant and to express this we define coercions between them: $t_1 \leq_T t_2$ may only hold if all terms of totality type t_1 also have totality type t_2 (assuming the underlying types are the same).

The relation $\leq_T$ is defined in Figure 3: it is reflexive, transitive, and anti-monotone in contravariant position. We write $\equiv$ for the equivalence induced by $\leq_T$, i.e. $t_1 \equiv t_2$ if and only if $t_1 \leq_T t_2$ and $t_2 \leq_T t_1$. The rule [top1] expresses that the totality type $!^{\top}\mathrm{ut}$ is the greatest among the totality types with the underlying type ut. One axiom derived from

$$[\text{ref}]\ \frac{}{t \leq_T t} \qquad [\text{trans}]\ \frac{t_1 \leq_T t_2 \quad t_2 \leq_T t_3}{t_1 \leq_T t_3}$$

$$[\rightarrow]\ \frac{t_1' \leq_T t_1 \quad t_2 \leq_T t_2'}{t_1 \rightarrow t_2 \leq_T t_1' \rightarrow t_2'} \qquad [\text{top1}]\ \frac{}{t \leq_T\ !^{\top}\varepsilon(t)}$$

$$[\text{top2}]\ \frac{}{!^{\top}(ut_1 \rightarrow ut_2) \leq_T\ !^{\top}ut_1 \rightarrow\ !^{\top}ut_2}$$

$$[\text{bot}]\ \frac{}{!^{\mathbf{b}}(ut_1 \rightarrow ut_2) \leq_T\ !^{\top}ut_1 \rightarrow\ !^{\mathbf{b}}ut_2}$$

$$[\text{notbot}]\ \frac{}{!^{\mathbf{n}}ut_1 \rightarrow\ !^{\mathbf{n}}ut_2 \leq_T\ !^{\mathbf{n}}(ut_1 \rightarrow ut_2)}$$

$$[\text{monotone}]\ \frac{}{t_1 \rightarrow t_2 \leq_T t_1' \rightarrow t_2'}\ \text{if } t_1' = \downarrow t_1 \text{ and } t_2' = \downarrow t_2$$

Figure 3: Coercions between totality types

the rule [top1] is

$$!^{\top}ut_1 \rightarrow\ !^{\top}ut_2 \leq_T\ !^{\top}(ut_1 \rightarrow ut_2) \tag{1}$$

Axiom (1) then motivates rule [top2] because when combined they yield

$$!^{\top}(ut_1 \rightarrow ut_2) \equiv\ !^{\top}ut_1 \rightarrow\ !^{\top}ut_2$$

The left-hand side of the rule [bot] represents the functions without WHNF and the right-hand side represents all non-terminating functions; this also includes the functions without WHNF. The rule [notbot] says that functions that map terms with a WHNF to a term with WHNF are also included in the functions with a WHNF.

The rule [monotone] ensures that we live in a universe of monotone functions: if we know less about the argument to a function, then we should know less about the result as well. The formulation of this requires the function $\downarrow$ on totality types defined by

$$\begin{array}{rcl@{\qquad\qquad}rcl} \downarrow(!^{\mathbf{b}}ut) & = & !^{\mathbf{b}}ut & \downarrow(!^{\top}ut) & = & !^{\top}ut \\ \downarrow(!^{\mathbf{n}}ut) & = & !^{\top}ut & \downarrow(t_1 \rightarrow t_2) & = & t_1 \rightarrow \downarrow t_2 \end{array}$$

The idea behind $\downarrow$ is that $\downarrow t$ is the smallest type (in the sense of "containing" fewest elements) such that both $t \leq_T \downarrow t$ and $\text{BOT}_T(\downarrow t)$ hold; for the proof see [16].

The relation $\leq_T$ is sound but not complete. The soundness result is part of Lemma 6 below. For the lack of completeness consider the two totality types $!^{\mathbf{b}}$`Int` $\rightarrow\ !^{\mathbf{n}}$`Int` and $!^{\top}\texttt{Int} \rightarrow !^{\mathbf{n}}\texttt{Int}$. It must be the case that every term with the first type also has the second type and vice versa since the terms are monotonic. However, although we can infer $!^{\top}\texttt{Int} \rightarrow !^{\mathbf{n}}\texttt{Int} \leq_T !^{\mathbf{b}}\texttt{Int} \rightarrow !^{\mathbf{n}}\texttt{Int}$ it turns out that we cannot infer $!^{\mathbf{b}}\texttt{Int} \rightarrow !^{\mathbf{n}}\texttt{Int} \leq_T !^{\top}\texttt{Int} \rightarrow !^{\mathbf{n}}\texttt{Int}$ using the coercions of Figure 3. This can be remedied by introducing the rule [monotone2] below: first we define the function $\uparrow$ on totality types as follows:

$$
\begin{aligned}
\uparrow(!^{\mathbf{b}}\mathrm{ut}) &= !^{\top}\mathrm{ut} & \qquad \uparrow(!^{\top}\mathrm{ut}) &= !^{\top}\mathrm{ut}\\
\uparrow(!^{\mathbf{n}}\mathrm{ut}) &= !^{\mathbf{n}}\mathrm{ut} & \qquad \uparrow(t_1 \rightarrow t_2) &= t_1 \rightarrow \uparrow t_2
\end{aligned}
$$

The idea behind $\uparrow$ is that it is the smallest type such that both $t \leq_T \uparrow t$ and $\mathrm{NOTBOT_T}(\uparrow t)$ hold where the predicate $\mathrm{NOTBOT_T}(t)$ must hold whenever the totality type must incorporate a term with a WHNF. Now we can write the new coercion rule for $\uparrow$:

$$[\text{monotone2}]\ \frac{}{t_1 \rightarrow t_2 \leq_T t_1' \rightarrow t_2'}\ \text{if } t_1' = \uparrow t_1 \text{ and } t_2' = \uparrow t_2$$

With this rule we can infer $!^{\mathbf{b}}\texttt{Int} \rightarrow !^{\mathbf{n}}\texttt{Int} \leq_T !^{\top}\texttt{Int} \rightarrow !^{\mathbf{n}}\texttt{Int}$. More work is needed to clarify if $\leq_T$ is complete with the new rule added.

Conjunction types

Based on the totality types we now define the conjunction types. A conjunction type, ct $\in$ CT, is either a totality type or a conjunction of two conjunction types:

$$\mathrm{ct} ::= t \mid \mathrm{ct} \wedge \mathrm{ct}$$

Thus conjunction is only allowed at the top-level (just like type-schemes in ML are only allowed at the top-level). The introduction of conjunction types means that it is possible to have empty types like $!^{\mathbf{n}}\texttt{Int} \wedge !^{\mathbf{b}}\texttt{Int}$. Actually, the fine details of empty types are closely connected with the choice of semantic model: emptiness of the type $(!^{\mathbf{b}}\texttt{Int} \rightarrow !^{\mathbf{n}}\texttt{Int} \rightarrow !^{\mathbf{n}}\texttt{Int}) \wedge (!^{\mathbf{n}}\texttt{Int} \rightarrow !^{\mathbf{b}}\texttt{Int} \rightarrow !^{\mathbf{n}}\texttt{Int}) \wedge (!^{\mathbf{b}}\texttt{Int} \rightarrow !^{\mathbf{b}}\texttt{Int} \rightarrow !^{\mathbf{b}}\texttt{Int})$ depends on whether the semantic model allows non-sequential behaviours of type $\texttt{Int} \rightarrow \texttt{Int} \rightarrow \texttt{Int}$. This will normally be the case for denotational semantics but will not be the case for natural-style operational semantics when the order of evaluation is forced (as when specifying lazy reduction to WHNF). The restriction to top-level conjunctions allows us to avoid some of the problems introduced by empty types; we return to this later.

A term can only have one underlying type; therefore a well-formed conjunction type will not involve types with different underlying types. The well-formedness predicate is defined by:

$$[\text{totality}]\ \frac{}{\vdash t}$$

$$[\wedge]\ \frac{\vdash \mathrm{ct}_1 \quad \vdash \mathrm{ct}_2}{\vdash \mathrm{ct}_1 \wedge \mathrm{ct}_2}\ \text{if } \varepsilon(\mathrm{ct}_1) = \varepsilon(\mathrm{ct}_2)$$

$$[\text{ref}]\ \frac{}{\text{ct} \leq_{CT} \text{ct}} \qquad [\text{trans}]\ \frac{\text{ct}_1 \leq_{CT} \text{ct}_2 \quad \text{ct}_2 \leq_{CT} \text{ct}_3}{\text{ct}_1 \leq_{CT} \text{ct}_3}$$

$$[\wedge 1]\ \frac{}{\text{ct}_1 \wedge \text{ct}_2 \leq_{CT} \text{ct}_1} \qquad [\wedge 2]\ \frac{}{\text{ct}_1 \wedge \text{ct}_2 \leq_{CT} \text{ct}_2}$$

$$[\wedge]\ \frac{\text{ct} \leq_{CT} \text{ct}_1 \quad \text{ct} \leq_{CT} \text{ct}_2}{\text{ct} \leq_{CT} \text{ct}_1 \wedge \text{ct}_2} \qquad [\text{type}]\ \frac{\text{t}_1 \leq_T \text{t}_2}{\text{t}_1 \leq_{CT} \text{t}_2}$$

Figure 4: Coercions between conjunction types

This allows us to overload the function ε to also find the underlying type of a conjunction type: $\varepsilon(\text{ct}_1 \wedge \text{ct}_2) = \varepsilon(\text{ct}_1)$. The predicate BOT_T is lifted to conjunction types:

$$\begin{aligned} \text{BOT}_{CT}(\text{ct}_1 \wedge \text{ct}_2) &= \text{BOT}_{CT}(\text{t}_1) \wedge \text{BOT}_{CT}(\text{t}_2) \\ \text{BOT}_{CT}(\text{t}) &= \text{BOT}_T(\text{t}) \end{aligned}$$

The rules for coercing between conjunction types are given in Figure 4.

The analysis

We have now prepared the ground for presenting the conjunction type inference system of Figure 5. The list A of assumptions gives totality types to free variables. Only the lambda abstraction can extend the assumption list and since conjunction types only can appear at the top-level this means that assumption lists always will associate totality types, not conjunction types, with the variables. For each constant c, we assume that a conjunction type ct_c is specified; as an example $\text{ct}_{\text{succ}} = (!^{\mathbf{n}}\texttt{Int} \rightarrow !^{\mathbf{n}}\texttt{Int}) \wedge (!^{\mathbf{b}}\texttt{Int} \rightarrow !^{\mathbf{b}}\texttt{Int})$.

The rules $[\text{var}]_T$, $[\text{abs}]_T$, $[\text{app}]_T$, and $[\text{const}]_T$ are just as their underlying type inference counterparts. There are three rules for conditional — depending on whether the test is of totality type $!^{\mathbf{b}}\texttt{Bool}$, $!^{\mathbf{n}}\texttt{Bool}$, or $!^{\top}\texttt{Bool}$.

The rule $[\text{coer}]_T$ can be applied to change the totality type to a greater totality type. It is quite useful as a preparation for applying rule [cond3]. The rule $[\text{conj}]_T$ allows to construct conjunction types (as is the case also for rule $[\text{const}]_T$).

From rule $[\text{fix}]_T$ we may derive rules

$$[\text{fix1}]_T\ \frac{A \vdash_T e : t \rightarrow t}{A \vdash_T \text{fix } e : t} \text{ if } \text{BOT}_T(t)$$

and

$$[\text{fix2}]_T\ \frac{A \vdash_T e : t_1 \rightarrow t_2}{A \vdash_T \text{fix } e : t_2} \text{ if } \text{BOT}_T(t_1) \text{ and } t_2 \leq_T t_1$$

$$[\mathrm{var}]_T \frac{}{A \vdash_T x : t} \text{ if } x : t \in A$$

$$[\mathrm{abs}]_T \frac{A, x : t_1 \vdash_T e : t_2}{A \vdash_T \lambda x.e : t_1 \rightarrow t_2}$$

$$[\mathrm{abs2}]_T \frac{A, x : t_1 \vdash_T e : t_2}{A \vdash_T \lambda x.e : !^{\mathbf{n}}\varepsilon(t_1 \rightarrow t_2)}$$

$$[\mathrm{app}]_T \frac{A \vdash_T e_1 : t_1 \rightarrow t_2 \quad A \vdash_T e_2 : t_1}{A \vdash_T e_1\ e_2 : t_2}$$

$$[\mathrm{cond1}]_T \frac{A \vdash_T e_1 : !^{\mathbf{b}}\mathtt{Bool} \quad A \vdash_T e_2 : ct \quad A \vdash_T e_3 : ct}{A \vdash_T \mathrm{cond}\ e_1\ e_2\ e_3 : !^{\mathbf{b}}\varepsilon(ct)}$$

$$[\mathrm{cond2}]_T \frac{A \vdash_T e_1 : !^{\mathbf{n}}\mathtt{Bool} \quad A \vdash_T e_2 : ct \quad A \vdash_T e_3 : ct}{A \vdash_T \mathrm{cond}\ e_1\ e_2\ e_3 : ct}$$

$$[\mathrm{cond3}]_T \frac{A \vdash_T e_1 : !^{\top}\mathtt{Bool} \quad A \vdash_T e_2 : ct \quad A \vdash_T e_3 : ct}{A \vdash_T \mathrm{cond}\ e_1\ e_2\ e_3 : ct} \text{ if } \mathrm{BOT}_{CT}(ct)$$

$$[\mathrm{fix}]_T \frac{A \vdash_T e : t_1 \rightarrow t_2 \wedge t_2 \rightarrow t_3 \wedge \ldots \wedge t_{n-1} \rightarrow t_n}{A \vdash_T \mathrm{fix}\ e : t_n} \text{ if } \begin{cases} \mathrm{BOT}_T(t_1), \\ \exists p, q : p < q \\ \wedge\ t_q \leq_T t_p, \end{cases}$$

$$[\mathrm{const}]_T \frac{}{A \vdash_T c : ct_C}$$

$$[\mathrm{coer}]_T \frac{A \vdash_T e : ct_1}{A \vdash_T e : ct_2} \text{ if } ct_1 \leq_{CT} ct_2$$

$$[\mathrm{conj}]_T \frac{A \vdash_T e : ct_1 \quad A \vdash_T e : ct_2}{A \vdash_T e : ct_1 \wedge ct_2}$$

Figure 5: Conjunction type Inference

that are simpler and more intuitive; they serve an important role in our proof strategy for the soundness result. Note that in rule $[\text{fix}]_T$ we have to ensure that the type t_1 can describe bottom in order to be able to calculate the fixpoint. After the first iteration the term has the totality type t_2 and after the second the totality type t_3, etc. When the term reaches the totality type t_q we can apply the rule $[\text{coer}]_T$ because we have $t_q \leq_T t_p$ and so the term has the totality type t_p. In this way we can go on as long as necessary to evaluate the fixpoint. Finally we iterate $n - q$ more times to get the type t_n for the fixpoint.

The following observations are easily verified: If we can infer $A \vdash_T e : ct$ then the conjunction type ct is well-formed; that is $\vdash ct$. The analysis is sound with respect to the underlying type system in the sense that if $A \vdash_T e : ct$ can be inferred, then so can $\varepsilon(A) \vdash_{UT} e : \varepsilon(ct)$. We also have a form of completeness: if we can infer $A \vdash_{UT} e : ut$ then we also have $\mathtt{top}(A) \vdash_T e : !^{\top}ut$ where $\mathtt{top}(x : ut, A) = x : !^{\top}ut, \mathtt{top}(A)$.

Example 3 In the system we can infer $\emptyset \vdash_T \text{fix } (\lambda x.x) : !^{\mathbf{b}}\mathtt{Int}$ which is more precise that the information obtained by [23] which in our notation is $!^{\top}\mathtt{Int}$. In the systems of [2, 8, 9] one can infer the type $!^{\top}\mathtt{Int}$ for the term fix $(\lambda x.7)$ whereas we infer $\emptyset \vdash_T \text{fix } (\lambda x.7) : !^{\mathbf{n}}\mathtt{Int}$ so again we are more precise. However, we cannot cope with the reordering of parameters: consider the term

$$\text{fix } (\lambda f.\lambda x.\lambda y.\lambda z.\text{cond } (z = 0)\ (x + y)\ (f\ y\ x\ (z\ \text{-}1)))$$

and the (well-formed) conjunction type

$$(!^{\mathbf{b}}\mathtt{Int} \to !^{\top}\mathtt{Int} \to !^{\top}\mathtt{Int} \to !^{\mathbf{b}}\mathtt{Int}) \wedge (!^{\top}\mathtt{Int} \to !^{\mathbf{b}}\mathtt{Int} \to !^{\top}\mathtt{Int} \to !^{\mathbf{b}}\mathtt{Int})$$

We *cannot* infer this type in our system because so far we only allow conjunction at the "top-level". The strictness analysis of [2, 8, 9] does not have this restriction on the use of conjunction types and may therefore obtain the desired type. □

4 Soundness

Our final task is to prove that the conjunction type inference system (Figure 5) is sound with respect to the natural-style operational semantics (Figure 2). First we define a predicate $\models e : ct$ stating that the term e is valid of conjunction type ct. Then we show some useful lemmas and finally we can prove the soundness result: if $A \vdash_T e : ct$ then $\models e[\overline{v}/\overline{x}] : ct$ for all closed substitutions $[\overline{v}/\overline{x}]$ that are valid of the types in A. For the full details of the proof see [16].

The validity predicate is shown in Figure 6. The term e is valid of conjunction type $ct_1 \wedge ct_2$ if e is valid of type ct_1 as well as ct_2. That the term e has a WHNF and the underlying type ut amounts to $\models e : !^{\mathbf{n}}ut$ being true; that e has no WHNF but has the underlying type ut amounts to $\models e : !^{\mathbf{b}}ut$ being true (i.e there exists no WHNF v such that $\vdash e \Downarrow v$). A term with conjunction type $!^{\top}ut$ just has to be of the underlying type

$$\text{(I)} \quad (\models e : ct_1 \wedge ct_2) \Leftrightarrow (\models e : ct_1) \wedge (\models e : ct_2)$$

$$\text{(II)} \quad (\models e : !^{\mathbf{b}}ut) \Leftrightarrow (\forall v: \not\vdash e \Downarrow v) \wedge (\emptyset \vdash_{UT} e : ut)$$

$$\text{(III)} \quad (\models e : !^{\mathbf{n}}ut) \Leftrightarrow (\exists v: \vdash e \Downarrow v) \wedge (\emptyset \vdash_{UT} e : ut)$$

$$\text{(IV)} \quad (\models e : !^{\top}ut) \Leftrightarrow (\emptyset \vdash_{UT} e : ut)$$

$$\text{(V)} \quad (\models e : t_1 \rightarrow t_2) \Leftrightarrow \left(\begin{array}{l} (\forall e' : (\models e' : t_1) \Rightarrow (\models e\, e' : t_2)) \wedge \\ (\emptyset \vdash_{UT} e : \varepsilon(t_1) \rightarrow \varepsilon(t_2)) \end{array} \right)$$

Figure 6: The definition of validity

ut, as we do not know anything about the evaluation of the term. A term e is valid of function type $t_1 \rightarrow t_2$ if for any other term e' that is valid of totality type t_1, also e applied to e' will be valid of totality type t_2.

Here we also see the importance of not having empty types; as with empty types the rule [notbot] will not be sound.

To prepare for the soundness of the conjunction type inference system we first need to bind all the free variables in the term. Let $\bar{x}$ be the list of variables in A, let $\bar{t}$ be the list of the totality types corresponding to the variables $\bar{x}$, and let $\bar{v}$ be a list of closed terms that are valid of the types $\bar{t}$, i.e. $\models \bar{v} : \bar{t}$. We now define $\models \bar{v} : \bar{t}$ inductively by

$$\begin{array}{rcl} \models (v, \bar{v}) : (t, \bar{t}) & = & (\models v : t) \wedge (\models \bar{v} : \bar{t}) \\ \models [\,] : [\,] & = & \mathtt{tt} \end{array}$$

The substitution $[\bar{v}/\bar{x}]$ is defined inductively by

$$\begin{array}{rcl} e[(v, \bar{v})/(x, \bar{x})] & = & (e[v/x])[\bar{v}/\bar{x}] \\ e[[\,]/[\,]] & = & e \end{array}$$

For the proof of soundness of the conjunction inference system we find it helpful to introduce the terms fix_n e where n is a number greater than or equal to 0. The idea is that n indicates how many times the fixpoint is allowed to be unfolded. So we need to expand the underlying type inference system and the semantics of the simply-typed λ-calculus. The underlying type of fix_n e is the same as for fix e:

$$[\mathrm{fix}_n]_{\mathrm{UT}} \frac{A \vdash_{UT} e : ut \rightarrow ut}{A \vdash_{UT} \mathrm{fix}_n\ e : ut}$$

and the semantics for fix_n e is:

$$[\mathrm{fix}_n]_{\mathrm{Sem}} \frac{\vdash e\ (\mathrm{fix}_n\ e) \Downarrow v}{\vdash \mathrm{fix}_{n+1}\ e \Downarrow v}$$

There are no rules for fix_0 e and hence fix_0 e is stuck. The underlying types that can be inferred for a term e without any fix_n's can also be inferred for the term e′ with fix_n replacing some occurrences of fix and vice versa. We do not allow the programmer to use fix_n; it is merely a piece of syntax needed to facilitate the proof of the soundness theorem.

Theorem 4 *Soundness* For expressions e without any fix_n we have
$(A \vdash_T e : ct) \Rightarrow (\forall \overline{v}: (\models \overline{v} : \overline{t}) \Rightarrow (\models e[\overline{v}/\overline{x}] : ct))$. □

Before we prove the soundness theorem we need some lemmas.

First we lift semantic equivalence to conjunction types:

Lemma 5 $((\models e_1 : ct) \wedge (e_1 \sim_{\varepsilon(ct)} e_2)) \Rightarrow (\models e_2 : ct)$ □

Proof By induction on ct. □

Next we note that our rules for $\leq_{CT}$ are sound:

Lemma 6 $((\models e : ct_1) \wedge (ct_1 \leq_{CT} ct_2)) \Rightarrow (\models e : ct_2)$ □

Proof By induction on the proof-tree for $ct_1 \leq_{CT} ct_2$. □

We know from the semantics that fix_0 e cannot evaluate hence it is valid of any type that can describe non-termination:

Lemma 7 $(\text{BOT}_T(t_1) \wedge \varepsilon(t_1) = \varepsilon(t_2) \wedge \models e : t_1 \rightarrow t_2) \Rightarrow (\models \text{fix}_0\ e : t_1)$ □

Proof It is easy to show that $\models \text{fix}_0\ e : !^{\mathbf{b}}\varepsilon(t_1)$ holds. Since we can show that $\text{BOT}_T(t_1)$ implies $!^{\mathbf{b}}\varepsilon(t_1) \leq_T t_1$ we obtain the result using Lemma 6. □

Unfolding fix_n or fix does not change validity:

Lemma 8 $(\models e\ (\text{fix}_n\ e) : ct) \Leftrightarrow (\models \text{fix}_{n+1}\ e : ct)$ □

Lemma 9 $(\models e\ (\text{fix}\ e) : ct) \Leftrightarrow (\models \text{fix}\ e : ct)$ □

Proof *of Lemma 8 and Lemma 9.* We show that fix_{n+1} e and e (fix_n e) are semantically equivalent, and then we apply Lemma 5. □

The relationship between fix_j and fix is clarified by:

Lemma 10 $(\exists j_0 : \forall j \geq j_0 : (\models \text{fix}_j\ e : t)) \Rightarrow (\models \text{fix}\ e : t)$ provided e is without any fix_n □

Proof By induction on t. □

Finally we can prove Theorem 4:

Proof *of Soundness Theorem* We assume that $A \vdash_T e : ct$ and that $\models \overline{v} : \overline{t}$ are true; we then prove $\models e[\overline{v}/\overline{x}] : t$ by induction on the proof-tree for $A \vdash_T e : ct$. Most of the cases are straightforward: we only give two of the cases.

The case [fix1]: We assume $A \vdash_T \text{fix } e : t$, $BOT_T(t) = \texttt{tt}$, and that $\models \overline{v} : \overline{t}$ is true. From the [fix1]-rule we get $A \vdash_T e : t \to t$ and by applying the induction hypothesis we have $\models e[\overline{v}/\overline{x}] : t \to t$. From Lemma 7 we get $\models \text{fix}_0\ e[\overline{v}/\overline{x}] : t$ and we now have $\models e[\overline{v}/\overline{x}]\ (\text{fix}_0\ e[\overline{v}/\overline{x}]) : t$. By applying Lemma 8 we have $\models \text{fix}_1\ e[\overline{v}/\overline{x}] : t$. We arrive at $\forall j \geq 0 : \models \text{fix}_j\ e[\overline{v}/\overline{x}] : t$ and we can apply Lemma 10 to get the result.

The case [fix2]: We assume $A \vdash_T \text{fix } e : t_2$, $BOT_T(t_1)$, $t_2 \leq_T t_1$, and that $\models \overline{v} : \overline{t}$ is true. From the [fix2]-rule we have $A \vdash_T e : t_1 \to t_2$ and by applying the induction hypothesis we get $\models e[\overline{v}/\overline{x}] : t_1 \to t_2$. Since $t_1 \to t_2 \leq_T t_1 \to t_1$ we have by Lemma 6 $\models e[\overline{v}/\overline{x}] : t_1 \to t_1$ and we can apply the proof of rule [fix1] to get $\models \text{fix } e[\overline{v}/\overline{x}] : t_1$. Now we have $\models e[\overline{v}/\overline{x}]\ (\text{fix } e[\overline{v}/\overline{x}]) : t_2$ and we can apply Lemma 9 to get the result.

□

5 Conclusion

We have described an inference system for combining strictness and totality analysis and we have proved the analysis sound with respect to a natural-style operational semantics. A promising approach towards the construction of an inference algorithm for strictness and totality types is to construct an abstract machine as suggested by Hankin and Le Métayer [6, 7]. We plan to investigate this in our future work and compare it with constraint based techniques.

We have briefly compared the results obtained by our analysis to those obtained by e.g. [2, 8, 9, 10, 23]. In some cases we get more precise results, in others they do. One may note that the type systems of Jensen [8] and Benton [2] allows general conjunction types. The reason that Jensen has no problems with unrestricted conjunctions is that it is not possible to construct empty types: the type system only includes the **b** and $\top$ annotated part of our system.

An open problem is the meaningful integration of lists and other data-types. For the strictness part one may be inspired by [21]. Consider the type A `list` where A is a base type. The totality type $(!^{\mathbf{n}}A)$`list` might then describe the finite lists with no bottom elements, the type $(!^{\mathbf{b}}A)$`list` might describe the infinite lists or lists with bottom elements, and the totality type $(!^{\top}A)$`list` might describe all list. The totality type of the map function would then be $(!^{\mathbf{n}}A \to !^{\mathbf{n}}B) \to (!^{\mathbf{n}}A)\texttt{list} \to (!^{\mathbf{n}}B)\texttt{list}$. Similarly, foldl and foldr will have totality types $(!^{\mathbf{n}}A \to !^{\mathbf{n}}B \to !^{\mathbf{n}}A) \to !^{\mathbf{n}}A \to (!^{\mathbf{n}}B)\texttt{list} \to !^{\mathbf{n}}A$ and $(!^{\mathbf{n}}A \to !^{\mathbf{n}}B \to !^{\mathbf{n}}B) \to !^{\mathbf{n}}B \to (!^{\mathbf{n}}A)\texttt{list} \to !^{\mathbf{n}}B$, respectively. However, to get this information from the analysis we need to analyse fixpoints in a better way, e.g. as suggested in [14].

Another open problem is to lift the restriction on the placement of conjunction; if successful, this will result in a somewhat more powerful system. One of the technical problems that need to be solved is the treatment of $\downarrow$ for conjunction types.

Acknowledgements Thanks to Olivier Danvy and Alan Mycroft for valuable discussions and to LOMAPS (Esprit Basic Research) and DART (Danish Science Research Council) for partial funding.

References

[1] Samson Abramsky. Abstract interpretation, logical relations and Kan extensions. *Journal of Logic and Computation*, 1(1):5–39, 1990.

[2] Nick Benton. *Strictness Analysis of Functional Programs*. PhD thesis, University of Cambridge, 1993. Available as Technical Report No. 309.

[3] Stefano Berardi. "Pruning" simply typed λ-terms. Technical report, Turin University, 1993.

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

[5] Philippa Gardner. Discovering needed reductions using type theory. In *Proceeding of TACS'94*, 1994.

[6] Chris Hankin and Daniel Le Métayer. Deriving algorithms from type inference systems: Application to strictness analysis. In *Proceedings of POPL'94*, pages 202 – 212, 1994.

[7] Chris Hankin and Daniel Le Métayer. Lazy type inference for the strictness analysis of lists. In *Proceedings of ESOP'94*, LNCS 788, pages 257 – 271, 1994.

[8] Thomas P. Jensen. Strictness analysis in logical form. In *Proceedings FPCA'91*, LNCS 523, pages 352 – 366, 1991.

[9] Thomas P. Jensen. Disjunctive strictness analysis. In *Proceedings LICS'92*, pages 174 – 185, 1992.

[10] Tsung-Min Kuo and Prateek Mishra. Strictness analysis: A new perspective based on type inference. In *Proceedings of FPCA'89*, pages 260 – 272. ACM Press, 1989.

[11] Alan Mycroft. The theory and practice of transforming call-by-need into call-by-value. In *Proceeding of the 4th International Symposium on Programming, LNCS 83*, pages 269–281, 1980.

[12] Alan Mycroft. *Abstract Interpretation and Optimising Transformation for Applicative programs*. PhD thesis, University of Edinburgh, Scotland, 1981.

[13] Alan Mycroft and Flemming Nielson. Strong abstract interpretation using power domain (extended abstract). In *Proceedings of ICALP'83*, LNCS 154, pages 536 – 547, 1983.

[14] Flemming Nielson, Hanne Riis Nielson. Termination Analysis. Manuscript, Aarhus University, 1994.

[15] David Sands. Complexity Analysis for a Lazy Higher-Order Language. In *Proceedings of ESOP'90*, LNCS 432, pages 361–376, 1990.

[16] Kirsten Lackner Solberg. Strictness and totality analysis. Forthcoming report, Odense University, Denmark, 1994.

[17] Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3):162 – 173, 1992.

[18] Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. In *Proceedings of LICS'92*, 1992.

[19] Mads Tofte and Jean-Pierre Talpin. Data region inference for polymorphic functional languages. In *Proceedings of POPL'94*, pages 188 – 201, 1994.

[20] D. A. Turner. Miranda: A non-strict functional language with polymorphic types. In *Proceeding of FPCA'85*, LNCS 201, pages 1 – 16, 1985.

[21] Phil Wadler. Strictness analysis on non-flat domains by abstract interpretation over finite domains. In S. Abramsky and C. Hankin (eds.), editors, *Abstract Interpretation of Declarative Languages*, pages 266 – 275. Ellis Horwood, 1987.

[22] Phil Wadler, John Hughes. Projections for strictness analysis. In *Proceedings of FPCA'87*, LNCS 274, 1987.

[23] David A. Wright. A new technique for strictness analysis. In *Proceedings of TAPSOFT'91*, LNCS 494, pages 260 – 272, 1991.