Refinement Types for Program Analysis

Mario Coppo, Ferruccio Damiani and Paola Giannini

Università di Torino, Dipartimento di Informatica, Corso Svizzera 185
10149 Torino (Italy)

Abstract. In this paper we introduce a system for the detection and elimination of dead code in typed functional programs. The main application of this method is the optimization of programs extracted from proofs in logical frameworks but it could be used as well in the elimination of dead code determined by program specialization. Our algorithm is based on a type inference system suitable for reasoning about dead code information. This system relays on *refinement* types which allow to exploit the type structure of the language for the investigation of program properties. The detection of dead code is obtained via type inference, which can be performed in an efficient and complete way, by reducing it to the solution of a system of inequalities between type variables. A key feature of our method is that program analysis can be performed in a strictly incremental way. Even though the language considered in the paper is a simply typed λ-calculus we can generalize our approach to polymorphic languages like ML. Although focused on dead code elimination our type inference method can also be applied to the investigation of other program properties like binding time and strictness. Some hints on these applications are given.

Introduction

Types have been recognized as useful in programming languages because they provide a semantical (non context free) analysis of programs. Such analysis is usually incorporated in the compiling process, and is used on one side to check the consistency of programs and on the other to improve the efficiency of the code produced.

In addition to preventing run-time errors, type systems can be useful for characterizing run-time properties of programs. For instance intersection types, see [4] (and also [1]), in their full generality, provide a characterization of strong normalization. As a consequence of this their type checking is undecidable, and therefore cannot be used in the compiling process of functional languages.

Type systems tailored to specific analysis, such as strictness, totality, binding time analysis etc. have been introduced, see [17, 12, 5, 13, 18]. In this perspective types represent program properties and their inference systems are systems for reasoning formally about them. In this paper we keep a clear distinction between the type structure of the language (types in the usual sense) and the *refinement* types which represent, inside the type structure of the language, particular properties. This distinction is very useful in the design of inference algorithms.

A similar view is taken in [8], in which the authors attempt to unify the analysis, identifying a basic type structure that can be extended to incorporate the various analyses.

The attraction of the type based approach is the possibility of designing efficient algorithms to do the analysis, as compared to the classical approach of semantical analysis using abstract interpretation which is quite inefficient for higher order functions. Type based analyzers are based on an implicit representation of types, either via type inequalities, see [11], or via lazy (implicit) types, see [8]. In our approach we pursue the first approach, reducing the inference problem to the solution of a system of inequalities between atomic types. An interesting feature of our method is that it is naturally compositional.

Type analysis is also used in the area of program extraction from formal proof. The programs extracted from proofs are usually very inefficient, as they contain parts that are useless for the computation of the final result; they therefore require some sort of simplification. One of the more effective simplification techniques is "pruning", and has been developed by Berardi, see [2]. In this technique useless terms are discovered by analyzing the type of terms. Such terms are called "dead code". The method was improved in [3] with the use of type inclusion. With type inclusion an application is well typed if the argument has a type included in the input type of the corresponding function. There are some basic problems with the method of [3]. The optimization algorithm is rather difficult to understand and this makes its proof of correctness even more difficult to follow. The method presented in this paper seems much more self-evident. Moreover in Berardi's algorithm it is assumed that the input term is typed, and the input type information is used for the analysis. So there does not seem to be a clear way to generalize the analysis to untyped terms (or better to do type inference and "pruning" at the same time). In [16] another formalism to represent information about dead code, based on the idea of marking terms and types, is presented. However no marking algorithm is given. This approach seems related to ours.

In our paper we present an inference system for detecting "dead code", and an algorithm that simplifies λ-terms based on the system of [3]. Although our system is tailored for dead code analysis we will show in the last section of the paper how its basic ideas can be applied to the design of inference algorithms for the investigation of other kinds of program properties.

The inference system for dead code relies mostly on Berardi's ideas. The language we consider is a typed (à la Church) λ-calculus with constants for natural numbers, pairs and recursor. The types are then built from the type of natural numbers, nat, using arrow and cartesian product. Starting from a typed term we infer properties of the term. We call such properties refinement types. For the dead code analysis we need two refinements for the basic type nat. The first, δ corresponds to the idea that the value may be used, and so it could only be replaced with a term with the same behaviour (β-equal). The second, ω, corresponds to the fact that the value is not used, and so it does not matter what the term is (it could be any constant of the same type). We consider an order relation $\sqsubseteq$ on refinement types whose interpretation is that a type is more informative than another, in particular $\delta \sqsubseteq \omega$. These properties are propagated to higher types, for instance if a function of type $\mathrm{nat} \to \mathrm{nat}$ has the refinement types $\delta \to \omega$ or $\omega \to \omega$ then the whole term will not be used (and also any free assumption on which it depends). When we apply a function to an argument we require that the argument have a refinement type that allows its use in every occurrence of the parameter. So it must be more specific, $\sqsubseteq$, than the refinement type of the formal parameter. The soundness of this system and of the optimizing transformation that it induces is proved via a partial equivalence relation semantics of the refinement types, showing that the optimized programs are observationally equivalent to the original ones.

Let us first consider a simple example. Let $M = (\lambda x^{\mathrm{nat}}.\bar{3})P$ where $\bar{3}$ is the numeral 3 and P is a term of type nat. Since x is never used in the body of the lambda we can assign the refined type $\omega \to \mathrm{nat}$ to $\lambda x.\bar{3}$ so we discover that P is not used in the computation of M and could be replaced by any constant of the right type. The information about dead code can be propagated. Consider having a λ-calculus with pairs $\langle\ ,\ \rangle$ and projections π_1 and π_2. Let M be the term $(\lambda x.\pi_1\langle P, Q\rangle)N$ for some term P, Q, and N. Since the projection π_1 returns the first component of the pair the term Q is dead code. Moreover, assume that P does not contain any occurrence of x, in this case also N is not relevant to the computation of the final result. So M in a call by name language behaves like the term P, which is of course simpler.

The main difference between our approach and that of [2] and [3] is in the algorithm that finds the optimized version of a given term. The algorithm presented in these papers is a kind of "data flow" algorithm that analyzes a term by implicitly building a directed graph which represents the input-output relation between the subterms of a given term. In our approach instead we assign to each term a system of inequalities between refinement type variables which can be seen as representing the whole flow of input-output information of the term. This system has always a maximal solution corresponding to the most informative typing of the term. Detecting the best optimization of a term corresponds to looking for a solution that maximizes the number of ω's with the restriction that the whole term must be useful (cannot be refined to ω). Such a solution can be found in a time proportional to the number of atomic type occurrences in the complete typing of a term. The proof of correctness and completeness of the algorithm is rather easy. An important feature of our algorithm is that it is naturally compositional while that of [2] and [3] is not.

Our analysis is for a strongly normalizing language, so we did not consider termination issues. To apply the method to languages including a fixed point operator we need a richer model including some sort of termination order on the terms.

It would be easy to extend the algorithm to deal with Curry-style polymorphism, see [9]. So for instance one could think of doing both type and refinement type inference at the same time. From the algorithmic point of view this extension is quite easy. In this framework it is, however, more difficult to understand the system from a semantical point of view. We can also deal with Milner-style polymorphism, see [15]. That is the presence of the "let $x = N$ in M" construct. Here x in M can be assigned different types, all instances of the same type scheme. In this case we have to handle the refinements of the different types of N in M. The optimizations possible for N would be such that they eliminate code that is useless according to all the different refinements of N.

The underlying type system is not part of the analysis. So the technique could be applied to more informative type systems such as the decidable restrictions of intersection types, like the rank 2 restriction (see [14]) or the simple types of [6] and [7]. Of course having already, at the level of the type system, more information the same refinement type system would be more informative. We think that, maintaining the orthogonality of the two issues: typing and refining the type, we can get the best out of each one of them.

The technique presented can be extended to deal with other properties useful in program optimization, such as binding time analysis and strictness analysis.

The first section introduces the language we are dealing with and the refinement type assignment system along with its semantics. In the second section we introduce a code optimization based on refinement type information, in particular we show that a term and its optimized version are equivalent. In the third section we introduce an algorithm for refinement types inference. The algorithm is articulated in two phases. First given a term we define a set of inequalities whose solutions induce all the derivations of refinement types for the term. Then we give an algorithm that finds the maximum solution of a system of inequalities. In the last section we outline how the technique of refinement types can be used to study binding time and strictness analysis.

1 A Type Assignment for Proving Properties

The aim of this section is to introduce a typed functional language (basically a typed λ-calculus with cartesian product and arithmetic constants) and a type assignment system for deriving flow properties of typed terms. The set of types is defined assuming as unique basic type nat, the set of natural numbers.

Definition 1 (Types). *The language of* types *($\mathcal{T}$) is defined by the following grammar:* $R ::= \mathrm{nat} \mid R \to R \mid R \times R$.

Types are ranged over by R, S, ... Typed terms are defined from a set of typed *term constants*

$$\begin{aligned}\mathcal{K}_A = \{\, & 0^{\mathrm{nat}},\ \mathrm{succ}^{\mathrm{nat}\to\mathrm{nat}},\ \mathrm{rec}^{\mathrm{nat}\to R\to(\mathrm{nat}\to R\to R)\to R},\ \mathrm{it}^{\mathrm{nat}\to R\to(R\to R)\to R},\\ & \mathrm{case}^{\mathrm{nat}\to R\to(\mathrm{nat}\to R)\to R},\ \mathrm{if}^{\mathrm{nat}\to R\to R\to R} \qquad \mid R\in\mathcal{T}\}\ ,\end{aligned}$$

(ranged over by C^S), and a set $\mathcal{V}_A$ of typed *term variables* (ranged over by $x^R, y^S, \ldots$). The choice of the constants has been done in view of an application to the optimization of terms extracted from proofs. We write $\Gamma \vdash_{\mathcal{T}} M^R$ to mean that M is a typed term of type R whose free variables are among the variables in the context Γ. We use this notation since it allows to attach a type to all subterms of M. Note the difference with the more usual notation $M : R$ in which this is not possible. The set of well typed (well decorated) terms is defined by the rules of the following definition.

Definition 2 (Typed terms). *A typing statement is an expression $\Gamma \vdash_{\mathcal{T}} M^R$ where Γ is a* context, *i.e., a set of variables x^R containing all the free variables of M. The rules for term formation are the following:*

$$(\mathrm{Var})\ \Gamma/x \cup \{x^R\} \vdash_{\mathcal{T}} x^R \qquad (\mathrm{Con})\ \Gamma \vdash_{\mathcal{T}} C^R$$

$$(\to \mathrm{I})\ \frac{\Gamma/x \cup \{x^R\} \vdash_{\mathcal{T}} M^S}{\Gamma/x \vdash_{\mathcal{T}} (\lambda x^R.M^S)^{R\to S}} \qquad (\to \mathrm{E})\ \frac{\Gamma \vdash_{\mathcal{T}} M^{R\to S} \quad \Gamma \vdash_{\mathcal{T}} N^R}{\Gamma \vdash_{\mathcal{T}} (M^{R\to S} N^R)^S}$$

$$(\times \mathrm{I})\ \frac{\Gamma \vdash_{\mathcal{T}} {M_1}^{R_1} \quad \Gamma \vdash_{\mathcal{T}} {M_2}^{R_2}}{\Gamma \vdash_{\mathcal{T}} \langle {M_1}^{R_1}, {M_2}^{R_2}\rangle^{R_1\times R_2}} \qquad (\times \mathrm{E}_l)\ \frac{\Gamma \vdash_{\mathcal{T}} M^{R_1\times R_2}}{\Gamma \vdash_{\mathcal{T}} (\pi_l M^{R_1\times R_2})^{R_l}}\ l \in \{1,2\}\ .$$

Note that with this notation we explicitly mention in M the types assigned to all its subterms. In the following we often omit to write types which are understood.

As usual a substitution is a finite function mapping term variables in terms, denoted $[x_1 := N_1, \ldots, x_n := N_n]$, which respect the types, i.e., each $x_i^{R_i}$ is substituted with a term $N_i^{R_i}$ of the same type.

Let $\Lambda_{\mathcal{T}}$ be the set of all typed terms which are defined according to the previous rules, i.e., $\Lambda_{\mathcal{T}} = \{M^R \mid \Gamma \vdash_{\mathcal{T}} M^R$ for some basis $\Gamma\}$. We provide $\Lambda_{\mathcal{T}}$ with a standard operational semantics defined by a notion of reduction. Let $\to_\beta$ denote the usual β reduction relation and $\to_\pi$ denote the following reduction relation for pairs: $\pi_l\langle M_1, M_2\rangle \to_\pi M_l$, for $l \in \{1,2\}$.
The reduction $\to_C$ determined by the constants is defined by the clauses:

$$\begin{array}{ll}
\mathrm{rec}\ 0\ M\ F \to_C M & \mathrm{it}\ 0\ M\ F \to_C M \\
\mathrm{rec}\ (\mathrm{succ}\ n)\ M\ F \to_C F\ n\ (\mathrm{rec}\ n\ M\ F) & \mathrm{it}\ (\mathrm{succ}\ n)\ M\ F \to_C F\ (\mathrm{it}\ n\ M\ F) \\
 & \\
\mathrm{case}\ 0\ M\ F \to_C M & \mathrm{if}\ 0\ M\ N \to_C M \\
\mathrm{case}\ (\mathrm{succ}\ n)\ M\ F \to_C F\ n & \mathrm{if}\ (\mathrm{succ}\ n)\ M\ N \to_C N\ .
\end{array}$$

Let $\to_r$ denote the union of $\to_\beta$, $\to_\pi$ and $\to_C$, let $\to_r^*$ denote its reflexive and transitive closure, and let $=_r$ be the equivalence relation induced by $\to_r^*$. Note that every term in $\Lambda_{\mathcal{T}}$ is strongly normalizable.

The *closed term model* $\mathcal{M}$ of $\Lambda_{\mathcal{T}}$ is defined by interpreting each type R as the set of the equivalence classes of the relation $=_r$ on the closed terms of type R. Let $\mathbf{I}(R)$ denote the interpretation of type R in this model, and let $[M]$ denote the equivalence class of term M. An *environment* is a mapping $e : \mathcal{V}_\Lambda \to \bigcup_{R\in\mathcal{T}} \mathbf{I}(R)$ which respects types, i.e., such that, for each x^R, $e(x^R) \in \mathbf{I}(R)$. The interpretation of a term M in an environment e is defined in a standard way by: $[\![M]\!]_e = [M[x_1 := N_1, \ldots, x_n := N_n]]$, where $\{x_1, \ldots, x_n\} = \mathrm{FV}(M)$ and $[N_l] = e(x_l)$ $(1 \le l \le n)$.

We want to be able to represent more information about terms of a given type. If we consider the basic type nat, for instance, we want to represent the notion of terms of type nat whose evaluation will possibly be useful to get the final result of a computation from that of terms (of the same type) whose evaluation will certainly be useless to that aim. For instance take the term of type nat: $M = (\lambda x^{\mathrm{nat}}.Q^{\mathrm{nat}})P^{\mathrm{nat}}$, where x does not occur in Q. The evaluation of P will be useless to the evaluation of M. To this aim we define two *refinement* types of nat: δ and ω, which represent, respectively, the notion of values which are (possibly) necessary or (certainly) useless for the determination of the final value of a computation, i.e., we identify δ with *(possibly) live* and ω with *dead*. So we will assign, for instance, type δ to Q and type ω to P.

Refinement types are defined from $\{\delta, \omega\}$ following the type construction rules. So, for instance, $\omega \to \delta$ is a refinement type of nat $\to$ nat (we denote this with $\omega \to \delta :: \mathrm{nat} \to \mathrm{nat}$) which informally represent the set of all functions f which yield a useful output whenever applied to an argument which is not useful for the termination of this output, like $\lambda x.Q$ above. This means that $\omega \to \delta$ characterizes all constant function of type nat $\to$ nat. In general we write $\rho :: R$ to mean that ρ refines type R.

Definition 3 (Refinement types). *The language $\mathcal{R}$ of* refinement types *(*r-types *for short) and the* refinement relation :: *are defined by the following rules:*

$$(\mathrm{nat})\ \frac{\varphi \in \{\delta, \omega\}}{\varphi :: \mathrm{nat}} \qquad (\to)\ \frac{\rho :: R \quad \sigma :: S}{\rho \to \sigma :: R \to S} \qquad (\times)\ \frac{\rho :: R \quad \sigma :: S}{\rho \times \sigma :: R \times S}\ .$$

It it easy to see that each refinement type ρ refines a unique type R, denoted by $\mathbf{T}(\rho)$, i.e., we have $\rho :: \mathbf{T}(\rho)$. Moreover, if R is a type and $\varphi \in \{\delta, \omega\}$, let $\mathbf{R}_\varphi(R)$ denote the refinement type obtained from R by replacing each occurrence of the basic type nat by φ. We have obviously $\mathbf{R}_\varphi(R) :: R$.

ω-refinement types (*ω-r-types* for short) formalize the notion of not being relevant to the computation at higher types.

Definition 4 (ω-r-types). *The set $\mathcal{O}$ of ω-r-types is inductively defined by: $\omega \in \mathcal{O}$ and if $\rho, \sigma \in \mathcal{O}$ and $\tau \in \mathcal{R}$ then $\tau \to \sigma \in \mathcal{O}$ and $\rho \times \sigma \in \mathcal{O}$.*

We now introduce a notion of inclusion between r-types, denoted $\sqsubseteq$, that means to be less informative.

Definition 5 (Inclusion relation). *The* inclusion relation $\sqsubseteq$ *between refinement types is defined by the following rules:*

$$(\delta)\ \delta \sqsubseteq \delta \qquad\qquad (\omega)\ \frac{\sigma \in \mathcal{O} \quad \mathbf{T}(\rho) = \mathbf{T}(\sigma)}{\rho \sqsubseteq \sigma}$$

$$(\to)\ \frac{\rho_1 \sqsubseteq \rho_2 \quad \sigma_1 \sqsubseteq \sigma_2}{\rho_2 \to \sigma_1 \sqsubseteq \rho_1 \to \sigma_2} \qquad\qquad (\times)\ \frac{\rho_1 \sqsubseteq \rho_2 \quad \sigma_1 \sqsubseteq \sigma_2}{\rho_1 \times \sigma_1 \sqsubseteq \rho_2 \times \sigma_2}\ .$$

Note that all ω-r-types which refines the same type are considered equivalent with respect to the $\sqsubseteq$ relation. Moreover we have immediately that for all $\rho_1, \rho_2 \in \mathcal{R}$, $\rho_1 \sqsubseteq \rho_2$ implies $\mathbf{T}(\rho_1) = \mathbf{T}(\rho_2)$.

Refinement types are assigned to typed λ-terms by a set of type inference rules similar to these of the ML type inference system.

If x^R is a term variable of type R an assumption for x^R is an expression of the form $x^R : \rho$, or $x : \rho$ for short, where $\rho :: R$. A basis is a set Σ of assumptions, and we write $\Sigma :: \Gamma$ to mean that Σ contains an assumption $x^R : \rho$ only if x^R

is a variable in Γ. We will prove judgements of the form $\Sigma \vdash_{\mathcal{R}} M^{\rho}$ where M is a typed term in a context Γ, i.e., such that $\Gamma \vdash_{\mathcal{T}} M^{R}$, $\Sigma :: \Gamma$ and $\rho :: R$. Note that we omit to denote explicitly the types of terms. Note also that we do not follow the more usual notation $\Sigma \vdash_{\mathcal{R}} M : \rho$ since we are interested in keeping track of the r-types assigned to the subterms of M in a derivation.

The type assignment rules are similar to the rules for term formation with the only exception that we take into account the inclusion relation. Notice that the types of all the constants C can be described by $T_C[R]$, where $T_C[\,]$ is a type context (with some, possibly none, holes) and R is a type. For instance $T_0[\,] = \text{nat}$ and $T_{\text{if}}[\,] = \text{nat} \to [\,] \to [\,] \to [\,]$. With $\tau_C[\,]$ we denote the r-type context obtained from $T_C[\,]$ by replacing nat with δ. For instance $\tau_0[\,] = \delta$ and $\tau_{\text{if}}[\,] = \delta \to [\,] \to [\,] \to [\,]$. Let $\rho :: R$, then $\tau_C[\rho] :: T_C[R]$ is the r-type obtained from $T_C[R]$ by replacing nat with δ and R with ρ.

Definition 6 (Refinement type assignment system). *The rules for refined type assignment are the following:*

$$(\text{Var})\ \frac{\rho_1 \sqsubseteq \rho_2}{\Sigma/x \cup \{x : \rho_1\} \vdash_{\mathcal{R}} x^{\rho_2}} \qquad (\text{Con})\ \frac{\rho :: R \quad \tau_C[\rho] \sqsubseteq \sigma}{\Sigma \vdash_{\mathcal{R}} C^{\sigma}}$$

$$(\to \text{I})\ \frac{\Sigma/x \cup \{x : \rho\} \vdash_{\mathcal{R}} M^{\sigma}}{\Sigma/x \vdash_{\mathcal{R}} (\lambda x^{\rho}.M^{\sigma})^{\rho \to \sigma}} \qquad (\to \text{E})\ \frac{\Sigma \vdash_{\mathcal{R}} M^{\rho_1 \to \sigma} \quad \Sigma \vdash_{\mathcal{R}} N^{\rho_2} \quad \rho_1 \to \sigma \sqsubseteq \rho_2 \to \sigma}{\Sigma \vdash_{\mathcal{R}} (M^{\rho_1 \to \sigma} N^{\rho_2})^{\sigma}}$$

$$(\times \text{I})\ \frac{\Sigma \vdash_{\mathcal{R}} {M_1}^{\rho_1} \quad \Sigma \vdash_{\mathcal{R}} {M_2}^{\rho_2}}{\Sigma \vdash_{\mathcal{R}} \langle {M_1}^{\rho_1}, {M_2}^{\rho_2} \rangle^{\rho_1 \times \rho_2}} \qquad (\times \text{E}_l)\ \frac{\Sigma \vdash_{\mathcal{R}} M^{\rho_1 \times \rho_2}}{\Sigma \vdash_{\mathcal{R}} (\pi_l M^{\rho_1 \times \rho_2})^{\rho_l}}\ l \in \{1, 2\}\ .$$

If $\Sigma \vdash_{\mathcal{R}} M^{\rho}$ then M^{ρ} has written in it the r-types assigned to its subterms. We say that M^{ρ} is an *annotated* term. It is worth mentioning that, in the rule $(\to \text{E})$, the condition $\rho_1 \to \sigma \sqsubseteq \rho_2 \to \sigma$ is used instead of $\rho_2 \sqsubseteq \rho_1$. This is because if $\rho_1 \to \sigma$ is an ω-r-type M can take any argument. Note that, being $\vdash_{\mathcal{R}}$ an inference system, the same terms can have different annotations.

Notice that the previous system is equivalent to the one obtained by removing the use of inclusion in rules (Var), (Con), $(\to \text{E})$ and by adding an explicit inclusion rule. We chose the presentation of Definition 6 to get a syntax directed system which is more suitable to the definition of the type inference algorithm.

The functions $\mathbf{T}$ and $\mathbf{R}$ defined above can naturally be extended to annotated terms. $\mathbf{T}(M^{\rho})$ in particular is simply the term M in which all r-type annotations have been erased. Is is immediate to see that: $\Sigma \vdash_{\mathcal{R}} M^{\rho}$ implies $\mathbf{T}(\Sigma) \vdash_{\mathcal{T}} \mathbf{T}(M^{\rho})$, and $\Gamma \vdash_{\mathcal{T}} M^{R}$ implies, for $\varphi \in \{\delta, \omega\}$, $\mathbf{R}_{\varphi}(\Gamma) \vdash_{\mathcal{R}} \mathbf{R}_{\varphi}(M^{R})$. This last deduction being obtained without the use of the $\sqsubseteq$ relation.

We now introduce a notion of semantics for our type assignment system. We interpret each basic refinement type δ or ω as a partial equivalence relation (p.e.r. for short) over the interpretation of type nat, i.e., the set of equivalence classes of closed terms of type nat with respect to $=_r$. Let $\times$ denote the cartesian product of sets and $[M]$ denote the equivalence class of M in $=_r$.

Definition 7. *1. The interpretation* $[\![\rho]\!]$ *of an r-type is defined by:*

$$[\![\delta]\!] = \{\langle [n],[n]\rangle \mid [n] \in \mathbf{I}(\mathrm{nat})\} \qquad [\![\omega]\!] = \mathbf{I}(nat) \times \mathbf{I}(nat) \qquad [\![\rho \times \sigma]\!] = [\![\rho]\!] \times [\![\sigma]\!]$$
$$[\![\rho \to \sigma]\!] = \{\langle [M],[N]\rangle \mid \forall \langle [P],[Q]\rangle \in [\![\rho]\!].\langle [MP],[NQ]\rangle \in [\![\sigma]\!]\}\ .$$

2. By $\sim_\rho$ *we denote the equivalence relation* $[\![\rho]\!]$ *on* $\mathbf{I}(\mathbf{T}(\rho))$*. Two environments* e_1*,* e_2 *are* Σ*-related if and only if, for all* $x^\sigma \in \Sigma$*,* $e_1(x) \sim_\sigma e_2(x)$*.*
3. Let $\Sigma \vdash_{\mathcal{R}} M^\rho$ *and* $\Sigma \vdash_{\mathcal{R}} N^\rho$*. We write* $M \sim_\rho^\Sigma N$ *to mean that for all* e_1*,* e_2*, if* e_1 *and* e_2 *are* Σ*-related, then* $[\![M]\!]_{e_1} \sim_\rho [\![N]\!]_{e_2}$*.*

The $\sqsubseteq$ relation between refinement types corresponds to inclusion of p.e.r. . In fact: $\rho \sqsubseteq \sigma$ if and only if $[\![\rho]\!] \subseteq [\![\sigma]\!]$. Moreover, if $\rho :: R$ is an ω-r-type, then $[\![\rho]\!] = \mathbf{I}(R) \times \mathbf{I}(R)$, i.e., $[\![\rho]\!]$ is the p.e.r. which relates all pairs of elements of $\mathbf{I}(R)$.

We state now the main theorem for p.e.r. interpretation, which is standard (in various forms) in the literature. The proof of the following theorem is by induction of terms.

Theorem 1. *Let* $\Sigma \vdash_{\mathcal{R}} M^\rho$*. Then* $M \sim_\rho^\Sigma M$*.*

Let us now identify a subset of r-typings that assures a correct use of the optimization mapping introduced in the next section.

Definition 8 (Faithful refinement). $\Sigma \vdash_{\mathcal{R}} M^\rho$ *is a* faithful *refinement typing statement if* $\rho = \mathbf{R}_\delta(\mathbf{T}(\rho))$*, and for all* $x : \sigma \in \Sigma$*, if* $\sigma \notin \mathcal{O}$ *then* $\sigma = \mathbf{R}_\delta(\mathbf{T}(\sigma))$*.*

Let $(\mathcal{C}[\ ]_\Gamma^R)^S$ denote a typed context of type S with a hole of type R in it which (possibly) binds variables in Γ. If $\Gamma \vdash_{\mathcal{T}} M^R$ and $\Gamma \vdash_{\mathcal{T}} N^R$ we say that M and N are *observationally equivalent* ($M =_{\mathrm{obs}} N$) if for all closed contexts $(\mathcal{C}[\ ]_\Gamma^R)^{nat}$ we have $\mathcal{C}[M] =_r \mathcal{C}[N]$.

2 Dead Code Elimination

In this section we introduce an optimization mapping $\mathbf{W}$ that, given an annotated term M^ρ, defines an optimized version of it. Define $\mathbf{W}(M^\rho)$ to be the term obtained by replacing all maximal subterms of M which are assigned an ω-r-type $\sigma :: R$ by Ω^σ, and define $\mathbf{W}(\Sigma) = \{x : \rho \mid x : \rho \in \Sigma \text{ and } \rho \notin \mathcal{O}\}$, where Σ is a r-type assignment basis. We have immediately that if $\Sigma \vdash_{\mathcal{R}} M^\rho$ then $\mathbf{W}(\Sigma) \vdash_{\mathcal{R}} \mathbf{W}(M^\rho)$ and $\mathbf{W}(\Sigma) \subseteq \Sigma$.

Example 1. Let $\Gamma \vdash_{\mathcal{T}} M^R$ where $\Gamma = \{x^{\mathrm{nat}}\}$, $R = (\mathrm{nat} \to \mathrm{nat}) \to \mathrm{nat}$ and $M^R = \lambda f^{\mathrm{nat}\to\mathrm{nat}}.f((\lambda z^{\mathrm{nat}}.3)(f\,x))$.
It is easy to check that $\Sigma \vdash_{\mathcal{R}} M'^\rho$, where $\Sigma = \{x : \omega\}$, $\rho = (\delta \to \delta) \to \delta$ and $M'^\rho = (\lambda f^{\delta\to\delta}.(f^{\delta\to\delta}((\lambda z^\omega.3^\delta)^{\omega\to\delta}(f^{\omega\to\omega}x^\omega)^\omega)^\delta)^\delta)^\rho$, is a faithful r-typing.
Applying the $\mathbf{W}$ optimization mapping we get $\mathbf{W}(\Sigma) \vdash_{\mathcal{R}} \mathbf{W}(M'^\rho)$, where $\mathbf{W}(\Sigma) = \{\}$ and $\mathbf{W}(M'^\rho) = (\lambda f^{\delta\to\delta}.(f^{\delta\to\delta}((\lambda z^\omega.3^\delta)^{\omega\to\delta}\Omega^\omega)^\delta)^\delta)^\rho$, and, erasing the r-type annotations, $\mathbf{T}(\mathbf{W}(\Sigma)) \vdash_{\mathcal{T}} \mathbf{T}(\mathbf{W}(M'^\rho))$, where $\mathbf{T}(\mathbf{W}(\Sigma)) = \{\}$ and $\mathbf{T}(\mathbf{W}(M'^\rho)) = \lambda f^{\mathrm{nat}\to\mathrm{nat}}.f((\lambda z^{nat}.3)\Omega^{\mathrm{nat}})$. □

The following result follows easily from the r-type semantics.

Theorem 2. *If $\Sigma \vdash_{\mathcal{R}} M^{\rho}$ then* $\mathbf{T}(M^{\rho}) \sim_{\rho}^{\Sigma} \mathbf{T}(\mathbf{W}(M^{\rho}))$.

This means that if $\Sigma \vdash_{\mathcal{R}} M^{\rho}$ then M and its optimized version are equivalent in ρ. The function $\mathbf{T}$ has been inserted just to point out that the r-type information has a purely static nature and is not relevant in the formation rules and in the operational semantics of terms.

This result is especially interesting when the typing of M is faithful since, using the above theorem, we can prove that that if $\Sigma \vdash_{\mathcal{R}} M^{\rho}$ is a faithful typing statement then $\mathbf{T}(M^{\rho})$ and $\mathbf{T}(\mathbf{W}(M^{\rho}))$ are observationally equivalent.

Theorem 3. *Let $\Sigma \vdash_{\mathcal{R}} M^{\rho}$ be a faithful typing. Then* $\mathbf{T}(M^{\rho}) =_{\text{obs}} \mathbf{T}(\mathbf{W}(M^{\rho}))$.

Remark 1. A strong optimization function $\mathbf{S}$ could be inductively defined on terms in such a way that, if $\Sigma \vdash_{\mathcal{R}} M'^{\rho}$ is the faithful r-typing of Example 1, then $\mathbf{T}(\mathbf{S}(\Sigma)) = \{\}$ and $\mathbf{T}(\mathbf{S}(M'^{\rho})) = \lambda f^{\text{nat}\to\text{nat}}.f\,3$.
The mapping $\mathbf{S}$ could be defined following [2]. The optimizations performed by $\mathbf{S}$ can be much stronger than those performed by $\mathbf{W}$. In certain cases $\mathbf{S}$ can also replace the constants of the language transforming, for instance, a recursor $\text{rec}^{\delta\to\rho\to(\omega\to\rho\to\rho)\to\rho}$ in an iterator $\text{it}^{\delta\to\rho\to(\rho\to\rho)\to\rho}$. A deeper analysis of the r-type structure of the term can allow to detect where this stronger optimization can be done. We do not go into those details since this kind of analysis simply uses the r-type structure determined by our algorithm. $\square$

3 An Algorithm for Refinement Types Inference

In this section we deal with the problem of defining an algorithm for program optimization based on the system defined in the previous section. To this aim the main problem is to use the inference rules to detect the maximal subterms to which ω-r-types can be assigned. The application of the optimization function $\mathbf{W}$ is then trivial. The algorithm, given a typing of a term, returns a decoration of the term with refinement patterns and a set of inequalities between refinement variables. The output of the algorithm characterizes all the possible r-typings of the term.

We start by defining the notions of r-type pattern and r-type scheme.

Definition 9 (Refinement type schemes).

1. *Let $\mathcal{U}$ be the set of* atomic variables, *ranged by α, β, γ, ...*
   The language $\mathcal{P}$ of refinement type patterns *(*r-patterns *for short) is defined from the rules of Definition 3 by replacing rule* (nat) *by the following rule:*
$$(\mathcal{U})\ \frac{\alpha \in \mathcal{U}}{\alpha :: \text{nat}}\quad ,$$
   r-patterns are ranged over by θ, η,...
2. *A* constraint *is a formula of one of the following shapes:*

– $\zeta_1 \sqsubseteq \zeta_2$, *where* $\zeta_1, \zeta_2 \in \{\delta\} \cup \mathcal{U}$
– $\mathcal{G} \Rightarrow \mathcal{E}$, *where* $\mathcal{G}$ *is a finite not empty subset of* $\{\delta\} \cup \mathcal{U}$ *and* $\mathcal{E}$ *is a finite set of constraints.*

3. *A* refinement type scheme *is a pair* $\langle \theta, \mathcal{E} \rangle$ *where* θ *is a r-pattern and* $\mathcal{E}$ *is a finite set of constraints.*

R-types and r-typings can be obtained from patterns by instantiation. A constraint is simply an inequality (between atomic variables or the constant δ) or a guarded set of constraints. For instance the set of constraints

$$\{\ \alpha_3 \sqsubseteq \alpha_1,\ \{\alpha_1, \alpha_2\} \Rightarrow \{\alpha_3 \sqsubseteq \alpha_4, \alpha_5 \sqsubseteq \delta\}\ \}$$

can be read as "$\alpha_3 \sqsubseteq \alpha_1$ and if $\alpha_1 = \delta$ or $\alpha_2 = \delta$, then $\alpha_3 \sqsubseteq \alpha_4$ and $\alpha_5 \sqsubseteq \delta$". To give a meaning to constraints we have to say which are the solutions of a set of constraints.

Definition 10 (Renaming and instantiations).

1. *A* renaming *is a one–to–one mapping* $r : \mathcal{U} \to \mathcal{U}$.
2. *An* instantiation *is a mapping* $i : \mathcal{U} \to \{\delta, \omega\}$.

Both renaming and instantiation can be extended to constants by defining $i(\varphi) = \varphi$ and $r(\varphi) = \varphi$, for $\varphi \in \{\delta, \omega\}$.

Definition 11. *Let* $\langle \theta, \mathcal{E} \rangle$ *be a scheme. An instantiation* i *satisfies* $\mathcal{E}$ *if*

– $\zeta_1 \sqsubseteq \zeta_2 \in \mathcal{E}$ *implies* $i(\zeta_1) \sqsubseteq i(\zeta_2)$, *and*
– $\mathcal{G} \Rightarrow \mathcal{E}' \in \mathcal{E}$ *implies that, if* $\delta \in i(\mathcal{G})$, *then* i *satisfies* $\mathcal{E}'$.

The set of all the instantiations that satisfy $\mathcal{E}$ is denoted by $\mathrm{SAT}(\mathcal{E})$. A scheme $\langle \theta, \mathcal{E} \rangle$ represents all the refinement types $i(\theta)$, for any $i \in \mathrm{SAT}(\mathcal{E})$.

Definition 12. *Let* i_1, i_2 *be instantiations.* $i_1 \sqsubseteq i_2$ *means that, for all* $\alpha \in \mathcal{U}$, $i_1(\alpha) \sqsubseteq i_2(\alpha)$.

Let $\mathcal{E}$ be a finite set of constraints. The set $\mathrm{SAT}(\mathcal{E})$ is not empty and has a maximum element.

Example 2. Consider the sets of constraints:

$$\begin{array}{ll}
\mathcal{E} = \{\ \{\alpha'_3\} \Rightarrow \{\alpha_3 \sqsubseteq \alpha'_3, \alpha'_2 \sqsubseteq \alpha_2\}, & \\
\quad\ \ \{\alpha''_3\} \Rightarrow \{\alpha_3 \sqsubseteq \alpha''_3, \alpha''_2 \sqsubseteq \alpha_2\}, & \mathcal{E}_1 = \{\ \{\alpha_1\} \Rightarrow \{\alpha_1 \sqsubseteq \delta\}, \\
\quad\ \ \alpha_1 \sqsubseteq \alpha'_1, & \qquad \alpha_2 \sqsubseteq \delta, \\
\quad\ \ \{\alpha''_3\} \Rightarrow \{\alpha'_1 \sqsubseteq \alpha''_2\}, & \qquad \alpha_3 \sqsubseteq \delta, \\
\quad\ \ \{\alpha_5\} \Rightarrow \{\alpha''_3 \sqsubseteq \alpha_4\}, & \qquad \alpha'_3 \sqsubseteq \delta \qquad \} . \\
\quad\ \ \{\alpha'_3\} \Rightarrow \{\alpha_5 \sqsubseteq \alpha'_2\} \qquad \} , &
\end{array}$$

To find the maximum element i_0 of $\mathrm{SAT}(\mathcal{E} \cup \mathcal{E}_1)$ observe that from the last three constraints of $\mathcal{E}_1$ we get $i_0(\alpha_2) = i_0(\alpha_3) = i_0(\alpha'_3) = \delta$. Then from the first constraint of $\mathcal{E}$ we get $i_0(\alpha'_2) = \delta$, and finally from the last constraint of $\mathcal{E}$ we have $i_0(\alpha_5) = \delta$. Let $\mathcal{I} = \{\alpha_2, \alpha'_2, \alpha_3, \alpha'_3, \alpha_5\}$, then i_0 defined by: $i_0(\alpha) = \delta$ if $\alpha \in \mathcal{I}$ and $i_0(\alpha) = \omega$ otherwise, is the maximum instantiation in $\mathrm{SAT}(\mathcal{E} \cup \mathcal{E}_1)$. $\square$

R-type inference of a term is reduced to the solution of a finite set of constraints. A maximal instantiation then corresponds to the typing that shows the maximal amount of dead code. The algorithm for finding the maximal instantiation i that satisfies a finite set of constraints $\mathcal{E}$ is presented in natural semantics style using judgements $\mathcal{E} \leadsto \mathcal{I}$ where $\mathcal{I}$ is the set of atomic variables that *represents* i, i.e., such that $\alpha \in \mathcal{I}$ if and only if $i(\alpha) = \delta$. The idea is simply that of recognizing, following the inequalities, all the variables that are forced to represent δ. All other atomic variables are then replaced by ω in the maximal solution.

Definition 13 ("Natural semantics" rules for constraints solution).

$$\text{(AX)}\ \frac{}{\mathcal{E}\ ;\ \emptyset}\ \text{if no other rule can be applied}$$

$$\text{(ATOM)}\ \frac{\mathcal{E}[\delta/\alpha]\ ;\ \mathcal{I}}{\mathcal{E} \cup \{\alpha \sqsubseteq \delta\}\ ;\ \mathcal{I} \cup \{\alpha\}} \qquad \text{(GUARD)}\ \frac{\mathcal{E} \cup \mathcal{E}''\ ;\ \mathcal{I} \qquad \delta \in \mathcal{G}}{\mathcal{E} \cup \{\mathcal{G} \Rightarrow \mathcal{E}''\}\ ;\ \mathcal{I}}$$

It is easy to see that, given a finite set of constraints $\mathcal{E}$, we can find $\mathcal{I}$ such that $\mathcal{E} \leadsto \mathcal{I}$ in a time linear in the number of constraints which occur in $\mathcal{E}$.

Proposition 1. *Let $\mathcal{E}$ be a finite set of constraints. Then $\mathcal{E} \leadsto \mathcal{I}$ if and only if $\mathcal{I}$ represents the maximum of* $\mathrm{SAT}(\mathcal{E})$.

We can now proceed to define the refinement type inference algorithm. This algorithm is presented in the natural semantics style using judgements $\langle \Gamma, M^R \rangle \Longrightarrow \langle \Theta, M'^{\,\theta}, \mathcal{E} \rangle$ where $\Gamma \vdash_{\mathcal{T}} M^R$, Θ is a basis that associates to each term variable in Γ a pattern, and $M'^{\,\theta}$ is a term annotated with patterns. The natural semantics rules follow the inference rules of Definition 6 in a natural way. We will prove that $\langle \theta, \mathcal{E} \rangle$ is a scheme that represents exactly the refined types assignable to M^R. More precisely, for any Σ and $M''^{\,\rho}$ such that $\mathbf{T}(\Sigma) = \Gamma$ and $\mathbf{T}(M''^{\,\rho}) = M^R$, we have that $\Sigma \vdash_{\mathcal{R}} M''^{\,\rho}$ if and only if $\Sigma = i(\Theta)$ and $M''^{\,\rho} = i(M'^{\,\theta})$, for some i that satisfies $\mathcal{E}$.

To define the algorithm we need some preliminary notations. Let R be a type. By $\mathrm{fresh}(R)$ we denote a pattern of the same shape of R such that to each occurrence of any atom in R is associated a fresh atomic variable. For example: $\mathrm{fresh}(\mathrm{nat} \to \mathrm{nat}) = \alpha \to \beta$. For a basis Γ, $\mathrm{fresh}(\Gamma) = \{x : \mathrm{fresh}(R) \mid x^R \in \Gamma\}$.
The function vars maps an r-pattern α to its finite set of atomic variables.
The function tail, that maps r-patterns and r-types (not containing ω) to finite subsets of $\{\delta\} \cup \mathcal{U}$, is inductively defined by: $\mathrm{tail}(\zeta) = \{\zeta\}$ (for $\zeta \in \{\delta\} \cup \mathcal{U}$), $\mathrm{tail}(\theta \times \eta) = \mathrm{tail}(\theta) \cup \mathrm{tail}(\eta)$, and $\mathrm{tail}(\theta \to \eta) = \mathrm{tail}(\eta)$.
Let θ', θ'' be r-patterns or r-types (not containing ω) of the same shape, $\mathrm{flat}(\theta' \sqsubseteq \theta'')$ denotes the set of constraints inductively defined by

$$\begin{array}{l}
\mathrm{flat}(\zeta_1 \sqsubseteq \zeta_2) = \{\zeta_1 \sqsubseteq \zeta_2\},\ \ \textit{if}\ \zeta_1, \zeta_2 \in \{\delta\} \cup \mathcal{U} \\
\mathrm{flat}(\theta_1 \times \eta_1 \sqsubseteq \theta_2 \times \eta_2) = \mathrm{flat}(\theta_1 \sqsubseteq \theta_2) \cup \mathrm{flat}(\eta_1 \sqsubseteq \eta_2) \\
\mathrm{flat}(\theta'_1 \to \cdots \to \theta'_n \to \eta' \sqsubseteq \theta''_1 \to \cdots \to \theta''_n \to \eta'') = \\
\ \{\mathrm{tail}(\eta'') \Rightarrow (\mathrm{flat}(\eta' \sqsubseteq \eta'') \cup \bigcup_{1 \leq l \leq n} \mathrm{flat}(\theta''_l \sqsubseteq \theta'_l))\}, \\
\ \textit{where}\ \eta', \eta''\ \textit{are not arrow r-patterns or arrow r-types.}
\end{array}$$

For all instances i, $i(\theta) \sqsubseteq i(\eta)$ if and only if $i \in \mathrm{SAT}(\mathrm{flat}(\theta \sqsubseteq \eta))$.

Definition 14 ("Natural semantics" rules for refinement typings inference).

$$(\mathrm{VAR})\ \frac{\Theta = \mathrm{fresh}(\Gamma) \quad \theta_1 = \mathrm{fresh}(R) \quad \theta_2 = \mathrm{fresh}(R)}{\langle \Gamma/x \cup \{x^R\}, x^R\rangle \Longrightarrow \langle \Theta/x \cup \{x : \theta_1\}, x^{\theta_2}, \mathrm{flat}(\theta_1 \sqsubseteq \theta_2)\rangle}$$

$$(\mathrm{CON})\ \frac{\Theta = \mathrm{fresh}(\Gamma) \quad \eta = \mathrm{fresh}(T_C[R]) \quad \theta = \mathrm{fresh}(R)}{\langle \Gamma, C^{T_C[R]}\rangle \Longrightarrow \langle \Theta, C^{\eta}, \mathcal{E}\rangle}$$

where $\mathcal{E} = \mathrm{flat}(\tau_C[\theta] \sqsubseteq \eta)$

$$(\mathrm{ABS})\ \frac{\langle \Gamma/x \cup \{x^R\}, M^S\rangle \Longrightarrow \langle \Theta/x \cup \{x : \theta\}, M'^{\eta}, \mathcal{E}\rangle}{\langle \Gamma/x, (\lambda x^R.M^S)^{R\to S}\rangle \Longrightarrow \langle \Theta/x, (\lambda x^{\theta}.M'^{\eta})^{\theta\to\eta}, \mathcal{E}\rangle}$$

$$(\mathrm{APP})\ \frac{\begin{array}{c}\langle \Gamma, M^{R\to S}\rangle \Longrightarrow \langle \Theta_1, M'^{\theta_1\to\eta}, \mathcal{E}_1\rangle \quad \langle \Gamma, N^R\rangle \Longrightarrow \langle \Theta_2, N'^{\theta_2}, \mathcal{E}_2\rangle \\ \Theta_1 = \{x : r(\theta) \mid x : \theta \in \Theta_2\}\end{array}}{\langle \Gamma, (M^{R\to S} N^R)^S\rangle \Longrightarrow \langle \Theta_1, (M'^{\theta_1\to\eta} N'^{\theta_2})^{\eta}, \mathcal{E}\rangle}$$

where $\mathcal{E} = \mathcal{E}_1 \cup r(\mathcal{E}_2) \cup \{\mathrm{tail}(\eta) \Rightarrow \mathrm{flat}(\theta_2 \sqsubseteq \theta_1)\}$
and r *is a renaming of the atomic variables in* Θ_2

$$(\mathrm{PAIR})\ \frac{\begin{array}{c}\langle \Gamma, M_1^{R_1}\rangle \Longrightarrow \langle \Theta_1, M_1'^{\theta_1}, \mathcal{E}_1\rangle \quad \langle \Gamma, M_2^{R_2}\rangle \Longrightarrow \langle \Theta_2, M_2'^{\theta_2}, \mathcal{E}_2\rangle \\ \Theta_1 = \{x : r(\theta) \mid x : \theta \in \Theta_2\}\end{array}}{\langle \Gamma, \langle M_1^{R_1}, M_2^{R_2}\rangle^{R_1\times R_2}\rangle \Longrightarrow \langle \Theta_1, \langle M_1'^{\theta_1}, M_2'^{\theta_2}\rangle^{\theta_1\times\theta_2}, \mathcal{E}_1 \cup r(\mathcal{E}_2)\rangle}$$

where r *is a renaming of the atomic variables in* Θ_2

$$(\mathrm{PROJ}_l)\ \frac{\langle \Gamma, M^{R_1\times R_2}\rangle \Longrightarrow \langle \Theta, M'^{\theta_1\times\theta_2}, \mathcal{E}\rangle}{\langle \Gamma, (\pi_l M^{R_1\times R_2})^{R_l}\rangle \Longrightarrow \langle \Theta, (\pi_l M'^{\theta_1\times\theta_2})^{\theta_l}, \mathcal{E}\rangle}\ l \in \{1,2\}$$

The rules are in this form to make them as readable as possible. They generate inequalities that could easily be avoided in a real implementation. Correctness and completeness of the inference are expressed by the following theorem.

Theorem 4. *$\Gamma \vdash_{\mathcal{T}} M^R$ and $\langle \Gamma, M^R\rangle \Longrightarrow \langle \Theta, M'^{\theta}, \mathcal{E}\rangle$ imply*

1. *for all instantiations i, if i satisfies $\mathcal{E}$, then $i(\Theta) \vdash_{\mathcal{R}} i(M'^{\theta})$*
2. *for all refinement type assignment statements $\Sigma \vdash_{\mathcal{R}} M''^{\rho}$ such that $\mathbf{T}(\Sigma) = \Gamma$ and $\mathbf{T}(M''^{\rho}) = M^R$ there exists $i \in \mathrm{SAT}(\mathcal{E})$ such that $i(\Theta) = \Sigma$ and $i(M'^{\theta}) = M''^{\rho}$.*

We are interested in faithful refinements, so we want to restrict the set of solutions of the constraints generated by the algorithm to those that correspond to faithful refinements. This can be done as shown by the following corollary.

Corollary 1. *Let $\Gamma \vdash_{\mathcal{T}} M^R$ and $\langle \Gamma, M^R\rangle \Longrightarrow \langle \Theta, M'^{\theta}, \mathcal{E}\rangle$. Then $i(\Theta) \vdash_{\mathcal{R}} i(M'^{\theta})$ is a faithful refinement type assignment if and only if the instantiations i satisfies the set of constraints $\mathcal{E} \cup \mathrm{faithful}(\Theta, \theta)$, where*
$\mathrm{faithful}(\Theta, \theta) = \bigcup_{x:\eta\in\Theta}\{\mathrm{tail}(\eta) \Rightarrow \{\gamma \sqsubseteq \delta | \gamma \in \mathrm{vars}(\eta)\}\} \ \cup\ \{\alpha \sqsubseteq \delta | \alpha \in \mathrm{vars}(\theta)\}$.

The constraint $\mathrm{tail}(\eta) \Rightarrow \{\gamma \sqsubseteq \delta | \gamma \in \mathrm{vars}(\eta)\}$ means that either η is an ω-r-type or it contains only δ's.

Let $\Gamma \vdash_{\mathcal{T}} M^R$ and $\langle \Gamma, M^R\rangle \Longrightarrow \langle \Theta, {M'}^{\theta}, \mathcal{E}\rangle$. If i_0 is the maximum element of $\mathrm{SAT}(\mathcal{E} \cup \mathrm{faithful}(\Theta, \theta))$ then, by the results of Sect. 1, $\mathbf{T}(\mathbf{W}(i_0({M'}^{\theta})))$ is the best simplification (in our sense) of M^R with the same operational meaning w.r.t. $=_{\mathrm{obs}}$.

Example 3. Let $\Gamma \vdash_{\mathcal{T}} M^R$ be the typing of Example 1. Then $\langle \Gamma, M^R\rangle \Longrightarrow \langle \Theta, {M'}^{\theta}, \mathcal{E}\rangle$ where $\Theta = \{x : \alpha_1\}$, $\theta = (\alpha_2 \to \alpha_3) \to \alpha'_3$,
${M'}^{\theta} = (\lambda f^{\alpha_2 \to \alpha_3}.(f^{\alpha'_2 \to \alpha'_3}((\lambda z^{\alpha_4}.3^{\alpha_5})^{\alpha_4 \to \alpha_5}(f^{\alpha''_2 \to \alpha''_3} x^{\alpha'_1})^{\alpha''_3})^{\alpha_5})^{\alpha'_3})^{\theta}$, and

$$\begin{aligned}\mathcal{E} = {} & \mathrm{flat}(\alpha_2 \to \alpha_3 \sqsubseteq \alpha'_2 \to \alpha'_3) \cup \mathrm{flat}(\alpha_2 \to \alpha_3 \sqsubseteq \alpha''_2 \to \alpha''_3) \cup \\ & \{\alpha_1 \sqsubseteq \alpha'_1, \{\alpha''_3\} \Rightarrow \{\alpha'_1 \sqsubseteq \alpha''_2\}, \{\alpha_5\} \Rightarrow \{\alpha''_3 \sqsubseteq \alpha_4\}, \{\alpha'_3\} \Rightarrow \{\alpha_5 \sqsubseteq \alpha'_2\}\}\end{aligned}$$

is the first set of constraints introduced in Example 2.

The set $\mathrm{faithful}(\Theta, \theta)$ is the set $\mathcal{E}_1$ in Example 2, so $\mathcal{E} \cup \mathrm{faithful}(\Theta, \theta) \rightsquigarrow \mathcal{I}$, where $\mathcal{I}$ is $\{\alpha_2, \alpha'_2, \alpha_3, \alpha'_3, \alpha_5\}$.

Let i_0 be defined by: $i_0(\alpha) = \delta$ if $\alpha \in \mathcal{I}$ and $i_0(\alpha) = \omega$ otherwise.
We have that $i_0(\Theta) \vdash_{\mathcal{R}} i_0({M'}^{\theta})$, where $i_0(\Theta) = \{x : \omega\}$, $i_0(\theta) = (\delta \to \delta) \to \delta$ and $i_0({M'}^{\theta}) = (\lambda f^{\delta \to \delta}.(f^{\delta \to \delta}((\lambda z^{\omega}.3^{\delta})^{\omega \to \delta}(f^{\omega \to \omega} x^{\omega})^{\omega})^{\delta})^{\delta})^{i(\theta)}$, is the faithful r-typing used in Example 1. □

When using the inference method to analyze a term M we will indeed produce as output a set of constraints that characterizes all possible r-typings of M. The subterms of M which are assigned ω-r-types in a faithful typing are indeed certainly useless in any use of M, and can then be simplified. Moreover the constraints associated to them are irrelevant to further analysis of M and can then be eliminated by the set of constraints for M. More precisely, if $\Gamma \vdash_{\mathcal{T}} M^R$, $\langle \Gamma, M^R\rangle \Longrightarrow \langle \Theta, {M'}^{\theta}, \mathcal{E}\rangle$ and i_0 is the maximum element of $\mathrm{SAT}(\mathcal{E} \cup \mathrm{faithful}(\Theta, \theta))$ then, the set of constraints $\mathcal{E}$ can be replaced by any set of constraints $\mathcal{E}'$ such that

$$\{i | i \in \mathrm{SAT}(\mathcal{E}') \text{ and } i_0 \sqsubseteq i\} = \{i | i \in \mathrm{SAT}(\mathcal{E}) \text{ and } i_0 \sqsubseteq i\} \ . \tag{1}$$

Example 4. Let $\mathcal{E}$ and i_0 be the set of constraints and the instantiation of Example 3. Consider the following subset $\mathcal{E}'$ of $\mathcal{E}$:

$$\mathcal{E}' = \{\{\alpha'_3\} \Rightarrow \{\alpha_3 \sqsubseteq \alpha'_3, \alpha'_2 \sqsubseteq \alpha_2\}, \{\alpha'_3\} \Rightarrow \{\alpha_5 \sqsubseteq \alpha'_2\}\} \ .$$

It is easy to see that any instantiation i' that maps to ω more atomic variables than i_0, i.e., such that $i_0 \sqsubseteq i'$, satisfies all the constraints in $\mathcal{E}$ that are not in $\mathcal{E}'$. So, since only the instantiations $i' \in \mathrm{SAT}(\mathcal{E})$ such that $i_0 \sqsubseteq i'$ are relevant to further analysis, we can replace $\mathcal{E}$ by the simplified set of constraint $\mathcal{E}'$ without loss of information. □

The following algorithm, given a finite set of constraints $\mathcal{E}$ and an instantiation $i_0 \in \mathrm{SAT}(\mathcal{E})$, returns a simplified set of constraints $\mathcal{E}'$ such that (1) holds, and so can be used to eliminate from $\mathcal{E}$ all the constraints which are not relevant for further analysis.
The algorithm is presented in natural semantics style using judgements $\mathcal{E}, \mathcal{I} \gg \mathcal{E}'$, where $\mathcal{I}$ represents i_0.

Definition 15 ("Natural semantics" rules for simplification of constraints).

$$(\text{AX})\ \frac{}{\mathcal{E},\mathcal{I} \gg \mathcal{E}} \text{ if no other rule can be applied}$$

$$(\text{ATOM}_1)\ \frac{\mathcal{E},\mathcal{I} \gg \mathcal{E}'}{\mathcal{E} \cup \{\delta \sqsubseteq \zeta\},\mathcal{I} \gg \mathcal{E}'} \qquad (\text{ATOM}_2)\ \frac{\mathcal{E},\mathcal{I} \gg \mathcal{E}' \quad \zeta_2 \notin \mathcal{I} \cup \{\delta\}}{\mathcal{E} \cup \{\zeta_1 \sqsubseteq \zeta_2\},\mathcal{I} \gg \mathcal{E}'}$$

$$(\text{GUARD}_1)\ \frac{\mathcal{E},\mathcal{I} \gg \mathcal{E}' \quad \mathcal{G} \cap (\mathcal{I} \cup \{\delta\}) = \emptyset}{\mathcal{E} \cup \{\mathcal{G} \Rightarrow \mathcal{E}_0\},\mathcal{I} \gg \mathcal{E}'} \quad (\text{GUARD}_2)\ \frac{\mathcal{E},\mathcal{I} \gg \mathcal{E}' \quad \mathcal{E}_0,\mathcal{I} \gg \emptyset}{\mathcal{E} \cup \{\mathcal{G} \Rightarrow \mathcal{E}_0\},\mathcal{I} \gg \mathcal{E}'}$$

$$(\text{GUARD}_3)\ \frac{\mathcal{E},\mathcal{I} \gg \mathcal{E}' \quad \mathcal{E}_0,\mathcal{I} \gg \mathcal{E}_0' \quad \mathcal{G} \cap (\mathcal{I} \cup \{\delta\}) \neq \emptyset \quad \mathcal{E}_0' \neq \emptyset}{\mathcal{E} \cup \{\mathcal{G} \Rightarrow \mathcal{E}_0\},\mathcal{I} \gg \mathcal{E}' \cup \{\mathcal{G} \Rightarrow \mathcal{E}_0'\}}$$

Proposition 2. *Let $\mathcal{E}$ be a finite set of constraints and let $\mathcal{I}$ represent $i_0 \in \text{SAT}(\mathcal{E})$. Then $\mathcal{E},\mathcal{I} \gg \mathcal{E}'$ implies that (1) holds.*

Example 5. Consider the output of the refinement type inference algorithm in Example 3, i.e., the triple $\langle \Theta, {M'}^{\theta}, \mathcal{E} \rangle$. The dead code showed by the maximum faithful typing can be immediately removed and, since $\mathcal{E},\mathcal{I} \gg \mathcal{E}'$, the set of constraints can be simplified as shown in Example 4. The triple $\langle \emptyset, M''^{\theta}, \mathcal{E}' \rangle$, where $M''^{\theta} = (\lambda f^{\alpha_2 \to \alpha_3}.(f^{\alpha_2' \to \alpha_3'}((\lambda z^{\alpha_4}.3^{\alpha_5})\Omega^{\alpha_3''})^{\alpha_5})^{\alpha_3'})^{\theta}$, and $\mathcal{E}' = \{\{\alpha_3'\} \Rightarrow \{\alpha_3 \sqsubseteq \alpha_3', \alpha_2' \sqsubseteq \alpha_2\}, \{\alpha_3'\} \Rightarrow \{\alpha_5 \sqsubseteq \alpha_2'\}\}$, is indeed all we need in the analysis of programs that use M. $\square$

We have now all the components for the definition of an efficient incremental analysis of a term M^R with free variables in Γ. Let $\langle \Gamma, M^R \rangle \Longrightarrow \langle \Theta, {M'}^{\theta}, \mathcal{E} \rangle$ and $\mathcal{E} \cup \text{faithful}(\Theta, \theta) \rightsquigarrow \mathcal{I}$. We can define an optimization mapping $\mathbf{W}'$ such that $\mathbf{W}'(\langle \Theta, {M'}^{\theta}, \mathcal{E} \rangle, \mathcal{I}) = \langle \Theta', {M''}^{\theta}, \mathcal{E}' \rangle$, where $\Theta' = \{x : \theta \mid x : \theta \in \Sigma \text{ and } \text{tail}(\theta) \cap \mathcal{I} \neq \emptyset\}$, M''^{θ} is obtained from M'^{θ} by replacing the maximal subterms annotated by a pattern η such that $\text{tail}(\eta) \cap \mathcal{I} = \emptyset$ by Ω^{η}, and $\mathcal{E},\mathcal{I} \gg \mathcal{E}'$. The triple $\langle \Theta', {M''}^{\theta}, \mathcal{E}' \rangle$ is all we need to perform further analysis of a program containing M. For instance M could be applied to a term or could be itself the argument of function. In such contexts further optimizations of M could be possible. The optimizations performed by $\mathbf{W}'$ are those which are possible in any context.

4 Binding Time and Strictness Analysis

The interpretation of δ and ω can be understood in a different way, as suggested in [10]. The r-type, δ (interpreted as the diagonal relation on $\mathbf{I}(\text{nat})$), characterizes values which are "known" and have a precise identity while ω characterizes values which are completely unknown (they can be any value in $\mathbf{I}(\text{nat})$). These properties are naturally propagated to higher types, for instance if a term of type $\text{nat} \to \text{nat}$ has the refinement types $\delta \to \omega$ or $\omega \to \omega$ then it is totally unknown, since it can be equated to anything, whereas the refinement $\delta \to \delta$ or $\omega \to \delta$ are given to terms that identify uniquely a function. In $\delta \to \delta$ two functions are equated only if they are extensionally equivalent. In the case of $\omega \to \delta$ we also know more, that is, the function is a constant function, since it identifies uniquely a value from an unknown input. Of course there are refinements that are not in either one of these classes, e.g., $(\omega \to \delta) \to \omega \to \delta$ that identifies all the functions that return a constant function when applied to a constant function. In general ω-r-types correspond to the concept of being totally unknown, indeed all of them represent the relations that identify all the elements of a given type.

With the previous interpretation detecting dead code in a term M can be seen as assuming that M identifies uniquely a value and trying to assign the maximum number of ω's to its subterms (of course keeping the refinement consistent). A subterm with ω-r-type then can be totally unknown, and so it is useless to the evaluation of the final result.

This interpretation led us to explore the use of the system for binding time analysis. For the binding time analysis the question is: given a description of the parameters of a function that will be known we must determine which parts of the program are dependent solely on the known parts. Known parameters correspond to terms identifying a unique value, and unknown parameters correspond naturally to terms having ω-r-types. The first are called static and the second dynamic in the binding time literature. For this analysis we look for a refinement of the term that respect the static and dynamic information of the parameters and that maximizes the number of δ in the term. A subterm of type δ is a subterm whose value does not depend on the values of the variables with dynamic type. So it is a natural candidate for evaluation when all static values are known. Our analysis gives that same results as [10] and [8].

Our framework can be also applied to strictness analysis. If we replace the rule (Zero) of Definition 6 by the rule "(Zero′) $\Sigma \vdash_{\mathcal{R}} 0^{\omega}$ " we get a system that can be used to study strictness properties for the λ-calculus with unary and constant operators. The refinement δ now means to be the undefined value of natural numbers and the refinement ω means to be any value. For a refinement of a functional type, say $\mathrm{nat} \to \mathrm{nat} \to \mathrm{nat}$, to specify that a function is strict in the first argument, means that $\delta \to \omega \to \delta$ is a refinement of the function. This is because if the first argument of the function is undefined and, regardless of the value of the second, the result is undefined that means that the first argument must be evaluated somewhere in the body of the function. In this context the equivalence of ω-r-types says that non-informative types are all equivalent. Of course strictness analysis makes sense in a language including a fixed point operator. It is not difficult to add to our language a typed fixed point constant, with a suitable refinement rule.

The difference between strictness analysis and dead code (or binding time) analysis arises with the use of constants such as $\mathrm{if}^{\mathrm{nat} \to R \to R \to R}$ which strictness behaviour cannot be described, as for dead code or binding time, by all the refinements σ such that $\delta \to \rho \to \rho \to \rho \sqsubseteq \sigma$ for any $\rho :: R$.
For strictness analysis, "if" has all the refinements σ such that either $\omega \to \rho \to \rho \to \rho \sqsubseteq \sigma$ or $\delta \to \rho_1 \to \rho_2 \to \rho_3 \sqsubseteq \sigma$. (The second case corresponding to a certainly undefined value of the test.) This adds a sort of and/or structure to the system of inequalities. A version of the system tailored to strictness analysis is in preparation.

References

1. H. P. Barendregt, M. Coppo, and M. Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *Journal of Symbolic Logic*, 48:931–940, 1983.
2. S. Berardi. Pruning Simply Typed Lambda Terms. *Journal of Symbolic Computation*, to appear.
3. S. Berardi and L. Boerio. Using Subtyping in Program Optimization. In *Typed Lambda Calculus and Applications*, 1995.
4. M. Coppo and M. Dezani-Ciancaglini. An extension of basic functional theory for lambda-calculus. *Notre Dame Journal of Formal Logic*, 21(4):685–693, 1980.
5. M. Coppo and A. Ferrari. Type inference, abstract interpretation and strictness analysis. In M. Dezani-Ciancaglini et al., editors, *A collection of contributions in honour of Corrado Böhm*, pages 113–145. Elsevier, 1993.
6. M. Coppo and P. Giannini. Pricipal Types and Unification for Simple Intersection Types Systems. *Information and Computation*, 122(1):70–96, 1995.
7. F. Damiani and P. Giannini. A Decidable Intersection Type System based on Relevance. In *Theoretical Aspects of Computer Software*, LNCS 789. Springer, 1994.
8. C. Hankin and D. Le Metayer. A Type-Based Framework for Program Analysis. In *Static Analisys*, LNCS 864, pages 380–394. Springer, 1994.
9. R. Hindley. The principal types schemes for an object in combinatory logic. *Transactions of American Mathematical Society*, 146:29–60, 1969.
10. L.S. Hunt and D. Sands. Binding Time Analysis: A New PERspective. In *Proceedings of the ACM Symposium on Partial Evaluation and Semantics-based Program Manipulation*, 1991.
11. P. O'Keefe J. Palsberg. A Type System Equivalent to Flow Analysis. In *Principles of Programming Languages*, 1995.
12. T. P. Jensen. Strictness Analysis in Logical Form. In J. Hughes, editor, *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, pages 98–105, 1991.
13. H. R. Nielson K. L. Solberg and F. Nielson. Strictness and Totality Analysis. In *Static Analisys*, LNCS 864, pages 408–422. Springer, 1994.
14. D. Leivant. Polymorphic Type Inference. In *Principles of Programming Languages*. ACM, 1983.
15. R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Science*, 17:348–375, 1978.
16. F. Prost. Marking techniques for extraction. Technical report, Ecole Normale Supérieure de Lyon, Lyon, December 1995.
17. T.M.Kuo and P.Mishra. Strictness analysis: a new perspective based on type inference. In *Functional Programming Languages and Computer Architecture*. ACM, 1989.
18. D. A. Wright. A New Technique for Strictness Analysis. In *Proceedings of TAPSOFT'91*, LNCS 494, pages 260–272. Springer, 1991.