Subtyping with Union Types, Intersection Types and Recursive Types II

Flemming Damm

N˚ 2259

May 1994

PROGRAMME 2

Calcul symbolique,

programmation

et génie logiciel

ISSN 0249-6399

Subtyping with Union Types, Intersection Types and Recursive Types II

Flemming Damm[*]

Programme 2 — Calcul symbolique, programmation et génie logiciel
Projet Lande

Rapport de recherche n ˚ 2259 — May 1994 — 23 pages

Abstract: This paper is a follow-up on previous work by the author on subtyping with (set-theoretic) union, intersection and recursive types. Previously, it was shown how types may be encoded as regular tree expressions/set constraints. This gave rise to a sound and complete decision procedure for type inclusion. The result was, however, limited to a rather specific type language.

In the work reported on here, we generalize the result and develop a general technique for deriving subtyping algorithms for type languages with union, intersection and recursive types. We present separate requirements for obtaining a subtyping algorithm which is respectively sound and complete. In this way we obtain a generic strategy for implementing the subtype relation for a broad class of very expressive type languages.

Key-words: Type theory, Regular tree expressions, Set constraints, Algorithms, Semantics, Ideal model, Intersection types, Union types, Recursive types

(Résumé : tsvp)

The author is partially funded by an HCM fellowship.

[*]Flemming.Damm@irisa.fr

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : (33) 99 84 71 00 – Télécopie : (33) 99 84 71 71

L'inclusion pour les types avec union, intersection et récursion

Résumé : Cet article poursuit les recherches de l'auteur sur l'inclusion de types avec union (ensembliste), intersection et récursion. Il a été montré aupararant comment les types pouvaient être représentés par des expressions d'arbres régulières (contraintes d'ensembles). Ceci a donné lieu à une procédure de décision correcte et complète pour l'inclusion de types. Cependant, le résultat était restreint à un langage de type assez spécifique.

Dans le travail présenté ici, nous généralisons le résultat et développons une technique générale pour dériver des algorithmes d'inclusion de types pour des langages de types avec union, intersection et récursion. Nous présentons une liste de conditions requises pour obtenir un algorithme d'inclusion de types correct et complet. De cette manière, nous obtenons une stratégie générique pour implémenter la relation d'inclusion de types pour une classe étendue de langages de types très expressifs.

Mots-clé : Théorie des types, Expressions d'arbres régulières, Contraintes d'ensembles, Algorithme, Sémantiques, Idéaux, Intersection de types, Union de types, Types récursifs

1 Introduction

1.1 Motivation

Type systems play an increasingly important role in the design of programming languages as well as in the implementation of compilers. Type systems have, e.g., been successfully applied in description of a broad class static analyses such as strictness analysis and tagging optimization.

Our work is motivated by the desire to make use of the expressiveness of (set-theoretic) union, intersection and recursive types. While recursive types are incorporated into many modern programming languages, union and intersection types have to the the knowledge of the author only been applied in a few experimental languages.

From a conceptual point of view, subtyping is a natural part of type systems including such constructs. However, the practical problems related with these constructs are many. To relate types properly, there are number of associativity, commutativity and distributivity rules which must be observed [Dam94a]. The two equalities

$$\begin{array}{rcl} \tau \to (\tau_1 \sqcap \tau_2) & = & (\tau \to \tau_1) \sqcap (\tau \to \tau_2) \\ (\tau_1 \sqcup \tau_2) \to \tau & = & (\tau_1 \to \tau) \sqcap (\tau_2 \to \tau) \end{array}$$

are examples of such rules. Here, the type $\tau_1 \to \tau_2$ denotes the set of functions which map all values of type τ_1 into values of type τ_2, and $\sqcup$ and $\sqcap$ denote respectively the union and intersection operator on types.

In addition to union, intersection and recursive types the type language considered in [Dam94a] contains flat, basic types, product types and function types. The work demonstrated how types may be encoded as regular tree expressions/set constraints. The basic idea is that types denote structurally similar values, and with an appropriate representation of these structures, types may be encoded as regular tree expressions. Since the inclusion problem for regular tree expressions is decidable, this makes up the basis of a decision procedure for the subtype relation. For the type language considered and its encoding, it is proved that this decision procedure is both sound and complete with respect to the ideal semantics of types.

However, most of the proofs are tightly bound to the specific language under consideration. It is neither clear how the results should be generalized, nor whether they are generalizable. Unfortunately, the language considered and its semantics subsume some very adverse limitations. Most crucial is an assumption that all types denote infinite sets of values.

As is usual when proving completeness results, a very precise demarcation of the problem is necessary. Since finite types obey very special subtyping rules, it comes as no surprise that types must be assumed infinite in order to prove completeness. Unfortunately, the proofs of respectively soundness and completeness are linked so closely together that the soundness result relies on the same limitations.

1.2 New Results

In this paper we reconsider the subtyping problem. The basic ideas are the same as in the first paper. The main result is a generic technique for implementing the subtype relation for a broad class of type languages. We deduce succinct requirements for the encoding of types into regular tree expressions. By reading these requirements as a guideline for defining an encoding of types, a general technique is obtained.

Type constructions to which our technique applies, include strict as well as lazy product and function types, record types, and types carrying control flow information.

As a byproduct we have a separate soundness proof for the encoding presented in [Dam94a]. This proof is liberated from the assumptions that all types must be infinite. Thus, for the considered sample language with function types, product types and basic types, we have a sound subtyping algorithm which is also complete when disregarding the rules applying specificly to finite types.

The results constitute the foundation for a further generalization to a technique for solving type inclusion constraints—the core of many type inference systems. This generalization is the topic of a forthcoming paper [Dam94b].

1.3 Related Work

Intersection types have been studied intensively in connection with the intersection type discipline [Hin92], but this type system has neither union types nor recursive types. Amadio and Cardelli [AC91] studied subtyping of recursive types. However, their techniques do not seem to be able to account for union and intersection types. Mishra and Reddy [MR85], Freeman and Pfenning [FP91], and Cartwright and Fagan [CF91] all defined type systems with union, intersection and recursive types. None of them did, however, consider the subtyping problem in its full generality.

The work by Aiken and Wimmers [AW93] comes very close to ours. They consider a type language similar to ours, and their work is also based on a relation of types and regular tree expressions. In fact they consider a more general problem: given a set of inclusion constraints, does there exists a substitution for the free variables such that the inclusions are valid. However, the underlying subtyping algorithm is strictly weaker than ours since rough approximations are introduced in connection with contravariant function types.

1.4 Overview

A short introduction to metric spaces is given in Section 2 where we also outline how a metric space of subsets of a given set may be defined. This is the core of both the interpretation of regular tree expressions which we present in Section 3, and the ideal model of types. A small type language with function types and product types is presented in Section 4. This language is equivalent to that of [Dam94a], and serves as an example on how our technique may be applied. Section 5 is the cornerstone of the exposition. Here we show how types may be encoded as regular tree expressions, present succinct requirements for the encoding of the basic type constructors, and give generic soundness and completeness proofs. In Section 6 we shortly demonstrate how some other type languages may be approached using our technique.

2 Metric Spaces

The theory of metric spaces is one of more theories which provide mathematical methods for giving semantics to recursive definitions, i.e. the theory is a fixed point theory. Some of the advantages associated with metric spaces are that fixed points are unique, and that the theory allows for finding fixed points of non-monotonic functions. The latter property is crucial for the semantics of types presented in Section 4.

Definition 2.1 (Metric spaces) *A* metric space (D, d) *is a set D together with a* metric $d : D \times D \to \mathbf{R}_+$ *subject to the conditions:*

(i) $d(x, y) = 0$ *iff* $x = y$

(ii) $d(x, y) = d(y, x)$

(iii) $d(x, z) \leq d(x, y) + d(y, z)$ *(The triangle property)*

A sequence $<s_i>_{i\geq 0}$ of elements of a metric (D, d) is called a *Cauchy sequence* if for every $\varepsilon > 0$ there exists an n such that for all $p, q \geq n$, $d(s_p, s_q) \leq \varepsilon$. A sequence $<s_i>_{i\geq 0}$ of elements is called *convergent* if there exists a limit $\hat{s} \in D$ of the sequence. That is, given any $\varepsilon > 0$, there exists an n such that for all $p \geq n$, $d(s_p, \hat{s}) \leq \varepsilon$. A metric space is *complete* if every Cauchy sequence is convergent.

A mapping $f : (D, d) \to (E, e)$ between two metric spaces is *contractive* if there exists a constant $c < 1$ such that for all $x, y \in D$,

$$e(f(x), f(y)) \leq c\, d(x, y)$$

The generalization to n-ary functions requires

$$e(f(x_1, \ldots, x_n), f(x'_1, \ldots, x'_n) \leq c \max\{d(x_i, x'_i) \mid i \leq n\}$$

A mapping is *non-expansive* if this requirement holds for $c \leq 1$. The *Banach fixed point theorem* states that contractive functions have unique fixed points:

Theorem 2.1 (Banach) *If (D, d) is a non-empty complete metric space and $f : (D, d) \to (D, d)$ is contractive, then f has a unique fixed point in (D, d). This fixed point is the limit of the Cauchy sequence $<f^i(x_0)>_{i\geq 0}$ where x_0 is an arbitrary point in D.*

For composition of contractive and non-expansive mappings we have:

Lemma 2.1 *Let $f : D_1 \times \ldots \times D_n \to E$ and $g_i : C \to d_i$ be arbitrary mappings. For the composition $f \circ <g_1, \ldots, g_n> = x \mapsto f(g_1(x), \ldots, g_n(x))$ we have*

1. *$f \circ \langle g_1, \ldots, g_n \rangle$ is non-expansive if f and all g_i are.*
2. *$f \circ \langle g_1, \ldots, g_n \rangle$ is contractive if f and all g_i are non-expansive and either f or all g_i are contractive.*

A mapping $f : (D, d) \rightarrow (E, e)$ between two metric spaces is continuous if for all $x \in D$ and all $\varepsilon > 0$ there exists a $\delta > 0$ such that for all $y \in D$, $d(x, y) < \delta \Rightarrow e(f(x), f(y)) < \varepsilon$. Continuous mappings preserve limits. That is, if $f : (D, d) \rightarrow (E, e)$ is a continuous mapping and $\langle s_i \rangle_{i \geq 0}$ is a convergent sequence in (D, d) with limit $\hat{s}$, then $\langle f(s_i) \rangle_{i \geq 0}$ is a convergent sequence in (E, e) with limit $f(\hat{s})$.

2.1 Metrics on Power Sets

Given any set D together with a rank function $r : D \rightarrow \mathbf{N}$, one can define a metric space of subsets of D as follows. Let s_1, s_2 be two sets, then $s_1 \ominus s_2$ denote their symmetric difference $(s_1 \setminus s_2) \cup (s_2 \setminus s_1)$. The closeness $c(s_1, s_2)$ of two subsets of D, $s_1, s_2 \in \mathcal{P}(D)$, is the element of least possible rank in the symmetric difference: $c(s_1, s_2) = \min\{r(b) \mid b \in s_1 \ominus s_2\}$. By convention, $\min\{\} = \infty$. The set $\mathcal{P}(D)$ of subsets of D constitutes a metric space with a metric d_r defined by: $d_r(s_1, s_2) = 2^{-c(s_1, s_2)}$. By convention, $2^{-\infty} = 0$. This space is complete:

Theorem 2.2 *The metric space $(\mathcal{P}(D), d_r)$ is complete. If $\langle s_i \rangle_{i \geq 0}$ is a Cauchy sequence then its limit is $\hat{s} = \{b \in D \mid b$ is in infinitely many $s_i\}$.*

It is worth noticing that this theorem is valid for an arbitrary rank function. In particular notice that the limit of a Cauchy sequence is independent of the actual rank function. A particular sequence may, however, fail to be a Cauchy sequence when the rank function is replaced.

Neither union, intersection nor complement are contractive as functions on a power set metric $(\mathcal{P}(D), d_r)$ but they are all non-expansive.

Let $f : D \times E_1 \times \ldots \times E_n \rightarrow D$ be a function of non-empty complete metric spaces. Assume f is contractive in its first argument. The "parameterized fixed point" function $\mu f : E_1 \times \ldots \times E_n \rightarrow D$ is defined by taking $(\mu f)(y_1, \ldots, y_n)$ to be the unique element $x \in D$ such that $x = f(x, y_1, \ldots, y_n)$. The Banach fixed point theorem ensures that such an x exists. In [MPS86] it is proved that if f is contractive (non-expansive) then so is μf. In connection with fixed points, we will not distinguish between $() \rightarrow D$ and D. I.e. the fixed point of $f : D \rightarrow D$ is $\mu f : () \rightarrow D$.

3 Regular Tree Expressions

Regular tree expressions [GS84, AM91] are conceptually equivalent to set constraints [AW92]. The difference is mainly syntactic. Given a ranked alphabet Σ, a Σ-tree as a labelled finite tree with node symbols Σ. For each node the number of subtrees is equal the rank of the symbol at that node. By H_Σ we denote the Herbrand universe of the symbols in Σ. The *height* $hg(t)$ of a tree t is the length of the longest path of t. A forest is a set of trees.

As an instance of the power set construction presented in Section 2.1, a complete metric space of forests may defined.

Theorem 3.1 *The metric space $(\mathcal{P}(H_\Sigma), d_{hg})$ is complete. If $<F_i>_{i \geq 0}$ is a Cauchy sequence then its limit is $\hat{F} = \{t \in H_\Sigma \,|\, t$ is in infinitely many $F_i\}$.*

Given a set of variables X, the set of *regular tree expressions* $\mathbf{RE}_0(\Sigma, X)$ is defined by the grammar:

$$\zeta ::= \mathsf{bot} \mid x \mid \sigma(\zeta, \ldots, \zeta) \mid \zeta + \zeta \mid \zeta * \zeta \mid \overline{\zeta} \mid \mathsf{fix}\, x.\zeta \tag{1}$$

For a constructor $\sigma \in \Sigma$ of rank n the number of arguments must be n. A regular tree expressions is *well-formed* if for all recursive regular tree expressions $\mathsf{fix}\, x.\zeta$, all occurrences of the variable x in ζ are embedded in constructions of the form $\sigma(\zeta_1, \ldots, \zeta_n)$. The set of well-formed regular tree expressions is denoted $\mathbf{RE}(\Sigma, X)$.

The semantic interpretation of regular tree expressions is provided by the function $\mathcal{I} : \mathbf{RE}(\Sigma, X) \to \mathbf{REEnv} \to \mathcal{P}(H_\Sigma)$ where $\mathbf{REEnv} = X \to \mathcal{P}(H_\Sigma)$ is the set of forest environments (ranged over by η) mapping variables into subsets of H_Σ.

$$\begin{array}{rcl}
\mathcal{I}[\![\mathsf{bot}]\!]\eta & = & \emptyset \\
\mathcal{I}[\![x]\!]\eta & = & \eta(x) \\
\mathcal{I}[\![\sigma(\zeta_1, \ldots, \zeta_n)]\!]\eta & = & \{\sigma(t_1, \ldots, t_n) \,|\, t_1 \in \mathcal{I}[\![\zeta_1]\!]\eta \wedge \ldots \wedge t_n \in \mathcal{I}[\![\zeta_n]\!]\eta\} \\
\mathcal{I}[\![\zeta_1 + \zeta_2]\!]\eta & = & \mathcal{I}[\![\zeta_1]\!]\eta \cup \mathcal{I}[\![\zeta_2]\!]\eta \\
\mathcal{I}[\![\zeta_1 * \zeta_2]\!]\eta & = & \mathcal{I}[\![\zeta_1]\!]\eta \cap \mathcal{I}[\![\zeta_2]\!]\eta \\
\mathcal{I}[\![\overline{\zeta}]\!]\eta & = & H_\Sigma \setminus \mathcal{I}[\![\zeta]\!]\eta \\
\mathcal{I}[\![\mathsf{fix}\, x.\zeta]\!]\eta & = & \mu(\lambda F \,.\, \mathcal{I}[\![\zeta]\!]\eta\{F/x\})
\end{array}$$

Well-formedness of regular tree expressions ensures that the application of the fixed point operator μ yields a well-defined result.

The set of trees corresponding to regular tree expression without free variables are strongly equivalent to tree recognizers [GS84]. Such sets are called regular.

Set constraints have been studied intensively the recent years. For our purpose, the important results are that for two regular tree expressions ζ_1 and ζ_2, it is both decidable whether $\mathcal{I}[\![\zeta_1]\!]\eta \subseteq \mathcal{I}[\![\zeta_2]\!]\eta$ for some forest environment η [AW92], and for all forest environments η [GTT93].

4 A Type Language

In this section we present a small type language. It it similar to that introduced in [Dam94a], and will serve as a sample language throughout of the rest of the paper. In addition to union, intersection and recursive types, it includes a set of basic types, product and function types.

In order to give semantics to types, we adopt the ideal model of types [MPS86]. This choice is essential for our development since many proofs relies on properties of this model.

However, the type inclusion rules applying in this model seem to be typical, and we expect them to be valid in most other models.

4.1 Syntax of Type Expressions

$$\tau ::= \bot \mid \iota_1 \mid \ldots \mid \iota_n \mid t \mid \tau \times \tau \mid \tau \to \tau \mid \tau \sqcup \tau \mid \tau \sqcap \tau \mid \mu t.\tau \tag{2}$$

The type $\bot$ denotes the empty type and ι_i are basic types. In the sequel we assume without loss of generality that all basic types are disjoint. Overlapping types may be defined in terms of the union type constructor. Products are non-associative. The function type is the well known contravariant function type. Union and intersection types are set-theoretic, and $\mu t.\tau$ denotes the unique type τ' fulfilling the equation $\tau' = \tau[\tau'/t]$. We assume all type expressions are *well-formed*: for all of recursive types $\mu t.\tau$, every occurrence of the variable t in τ occurs in the scope of a basic type constructor, i.e. in a product or function type. This requirement ensures unique fixed points for recursively defined types. By **TExp** we denote the set of well-formed type expressions.

A top type $\top$ which is the union of all types may be defined in terms of recursion:

$$\top \quad \equiv \quad \mu t.\iota_1 \sqcup \ldots \sqcup \iota_n \sqcup (t \times t) \sqcup (\bot \to t)$$

4.2 Semantic Domains

The value universe in which the types will be interpreted, is a domain $\mathbf{V}$ satisfying the domain equation:

$$\mathbf{V} = \mathbf{B}_1 + \ldots + \mathbf{B}_n + (\mathbf{V} \times \mathbf{V}) + (\mathbf{V} \to \mathbf{V}) \tag{3}$$

Here $\mathbf{B}_i$ denote flat domains of basic values. These domains are assumed disjoint. By $\mathbf{V} \times \mathbf{V}$, we denote the product of $\mathbf{V}$ with itself, and $\mathbf{V} \to \mathbf{V}$ is the continuous functions from $\mathbf{V}$ to $\mathbf{V}$. Here we have two parameters to the semantics. At first, the product $\mathbf{V} \times \mathbf{V}$ may be either the Cartesian product or the smashed product. The smashed product differs from the Cartesian product by coalescing a pair $<x, y>$ with the bottom element of the product whenever x or y equals $\bot$. The other parameter concerns the function domain. Here we may or may not include non-strict functions. For our example, we choose the smashed product and assume all functions strict. In this way, we are consistent with [Dam94a]. The domain operator $+$ denote the sum operator. Here we have a third parameter to the semantics. Some of the domains may be lifted. For example, one might want to lift the function domain to obtain a separate bottom value for the functions $\lambda x \,.\, \bot \neq \bot$. Again for consistency with [Dam94a] we assume that none of the summand domains are lifted.

It should be emphasized that the ideal model (and thus the rest of this section) is independent of these choices. However, as one may expect, the type inclusion rules depend on the instantiations of these parameters. Other instantiations give other subtyping rules. As an example, we have that $\top \to \bot = \bot$ with the choices made. But this equality is not valid if the function domain is lifted. We discuss the other possibilities in Section 6.

Solutions of domain equations like (3) exists up to isomorphism and may be constructed via the so-called inverse limit construction. The solution is a limit of the sequence of cpos $<\mathbf{V}_i>_{i\geq 0}$, starting from $\mathbf{V}_0 = \{\perp\}$:

$$\mathbf{V}_{i+1} = \mathbf{B}_1 + \ldots + \mathbf{B}_n + (\mathbf{V}_i \times \mathbf{V}_i) + (\mathbf{V}_i \rightarrow \mathbf{V}_i)$$

This sequence constitute a sequence of increasingly better approximations to the solution.

An element d of a domain D is *ω-finite* (or just finite) if for all increasing sequences $<x_n>_{n\geq 0}$ of elements of D, $d \sqsubseteq \bigsqcup_n x_n$ implies that there exists an n such that $d \sqsubseteq x_n$. The finite elements of a cpo D is denoted by D°. The finite elements of a product domain $D \times E$ is $D^\circ \times E^\circ$, and the finite elements of a sum domain $D + E$ is $D^\circ + E^\circ$. For a function domain $D \rightarrow E$ the finite elements are finite lubs of *step functions.* For $d \in D^\circ$ and $e \in E^\circ$, a step function $d \Rightarrow e$ is defined by:

$$(d \Rightarrow e)(x) = \begin{cases} e & \text{if } x \sqsupseteq d \\ \perp & \text{otherwise} \end{cases}$$

Notice that for any finite function value f there exists a minimal set of step functions such that f is the lub of these step functions. Given a set $\{a_1 \Rightarrow b_1, \ldots, a_n \Rightarrow b_n\}$ of step functions, the minimal set of step functions may be obtained by consecutively removing those step functions $a_i \Rightarrow b_i$ for which there exists another step function $a_j \Rightarrow b_j$ such that $b_j = b_i$ and $a_j \sqsubseteq a_i$. Whenever we in the following write a finite function value as a lub of a set of step functions, we assume this set to be minimal.

For a finite value $v \in \mathbf{V}^\circ$ we define the *rank* $r(v)$ to be the least i such that $v \in \mathbf{V}_i$. Informally speaking, a value is finite if its structure is finite, and one can think the rank of a value as the height its structure.

4.3 The Metric Space of Ideals

Order ideals are non-empty downward closed subsets of a partial order. For a partial order D the order ideals are denoted by $\mathcal{J}_0(D)$. An *ideal* is an order ideal which is closed under lubs of increasing sequences. For a partial order D the ideals are denoted by $\mathcal{J}(D)$.

As mentioned, we shall pay special attention to finite elements. The following theorem establishes a close connection between order ideals of finite elements and ideals:

Theorem 4.1 *The function $I \mapsto I^\circ$ mapping an ideal I into the corresponding set of finite values I° is an order isomorphism of $(\mathcal{J}(\mathbf{V}), \subseteq)$ and $(\mathcal{J}_0(\mathbf{V}^\circ), \subseteq)$.*

Using the previously defined rank function r on finite values, we get as a special instance of Theorem 2.2:

Theorem 4.2 *The metric space $(\mathcal{P}(\mathbf{V}^\circ), d_r)$ is complete. If $<I_i>_{i\geq 0}$ is a Cauchy sequence then its limit is $\hat{I} = \{v \in \mathbf{V} \mid v \text{ is in infinitely many } I_i\}$.*

For a Cauchy sequence $<I_i>_{i\geq 0}$ of order ideals the limit $\hat{I} = \{x \in \mathbf{V}^\circ \mid b \text{ is in infinitely many } I_i\}$ is an order ideal. Therefore, the subspace $\mathcal{J}_0(\mathbf{V}^\circ, d_r)$ is also complete and from Theorem 4.1 it follows that the metric space $(\mathcal{J}(\mathbf{V}), d_r)$ is complete as well.

Corresponding to the basic type constructors $\times$ and $\rightarrow$ we may define the functions:

$$\begin{array}{rcl} I \boxtimes J & = & \{<x,y> \in \mathbf{V} \times \mathbf{V} \mid x \in I \wedge y \in J\} \\ I \boxminus J & = & \{f \in \mathbf{V} \rightarrow \mathbf{V} \mid \forall a \in I \,.\, f(a) \in J\} \end{array}$$

They are both contractive in the metric space $(\mathcal{J}(\mathbf{V}), d_r)$. But this fact is highly sensitive to chosen rank function r.

4.4 Semantics of Types

Let $\mathbf{TEnv} = \mathbf{Var} \rightarrow \mathcal{J}(\mathbf{V})$ be the set of *proper* type environments (ranged over by ρ). The semantic function on type expressions $\mathcal{D} : \mathbf{TExp} \rightarrow \mathbf{TEnv} \rightarrow \mathcal{J}(\mathbf{V})$ mapping well-formed type expressions into ideals, is defined by:

$$\begin{array}{rcl} \mathcal{D}[\![\bot]\!]\rho & = & \{\bot\} \\ \mathcal{D}[\![\iota_i]\!]\rho & = & \mathbf{B}_i \\ \mathcal{D}[\![t]\!]\rho & = & \rho(t) \\ \mathcal{D}[\![\tau_1 \times \tau_2]\!]\rho & = & \mathcal{D}[\![\tau_1]\!]\rho \boxtimes \mathcal{D}[\![\tau_2]\!]\rho \\ \mathcal{D}[\![\tau_1 \rightarrow \tau_2]\!]\rho & = & \mathcal{D}[\![\tau_1]\!]\rho \boxminus \mathcal{D}[\![\tau_2]\!]\rho \\ \mathcal{D}[\![\tau_1 \sqcup \tau_2]\!]\rho & = & \mathcal{D}[\![\tau_1]\!]\rho \cup \mathcal{D}[\![\tau_2]\!]\rho \\ \mathcal{D}[\![\tau_1 \sqcap \tau_2]\!]\rho & = & \mathcal{D}[\![\tau_1]\!]\rho \cap \mathcal{D}[\![\tau_2]\!]\rho \\ \mathcal{D}[\![\mu t.\tau]\!]\rho & = & \mu(\lambda I \,.\, \mathcal{D}[\![\tau]\!]\rho\{I/t\}) \end{array}$$

Well-formedness of type expressions ensures that the semantics of a recursive type is well-defined and unique. This gives well-definedness of $\mathcal{D}$ as well. By $\mathcal{D}^\circ[\![\tau]\!]\rho$ we denote the finite values of $\mathcal{D}[\![\tau]\!]\rho$. We shall frequently consider the semantics of types in *improper* environments: a type environment ρ is improper if for some type variables t, $\rho(t)$ is a set of values which is not an ideals. As long as types do not contain recursion, $\mathcal{D}$ is well-defined for improper type environments.

4.5 The Metric Space of Ideals—Reconsidered

One of our results is a new soundness proof of the subtyping approach based on encoding of types as regular tree expressions. This proof does not assume finiteness of types. The proof includes a relation of the metric space of ideals with a metric space of sets of trees. In this connection it turns out that the metric defined in Section 4.3 causes certain problems. Fortunately, it is possible to redefine the metric without affecting the semantics of types.

The metric space of ideals is an instance of the subset construction presented in Section 2.1. As noticed in Section 2.1, the completeness of such spaces is independent of the actual rank function on the underlying set. Consequently, we can redefine the rank function r on $\mathbf{V}^\circ$ without affecting the completeness of the metric space of ideals. In particular, the limit of a Cauchy sequence is independent of the actual rank function. Thus, as long as the semantic function $\mathcal{D}$ is well-defined, it is independent of r. For well-definedness of $\mathcal{D}$ the decisive factor is that for all recursive types $\mu t.\tau$ the sequence $<D^i(I_0)>_{i\geq 0}$ with $D \equiv \lambda I\,.\,\mathcal{D}[\![\tau]\!]\rho\{I/t\}$ must be a Cauchy sequence for all ideals I_0 and type environments ρ.

The rank function r' defined below serves our purposes:

$$\begin{array}{rcl} r'(\bot) & = & 0 \\ r'(b_{ij}) & = & 1 \\ r'(<x,y>) & = & 1+\max\{r'(x), r'(y)\} \\ r'(a_1 \Rightarrow b_1 \sqcup \ldots \sqcup a_n \Rightarrow b_n) & = & n+1+\max\{r'(a_1), r'(b_1), \ldots, r'(a_n), r'(b_n)\} \end{array}$$

Here we assume that a finite function value is described as the lub of a minimal set of step functions. This ensures well-definedness of r'.

The important difference between r and r' is in the definition of the rank of a function value. For the old rank function r we have:

$$r(a_1 \Rightarrow b_1 \sqcup \ldots \sqcup a_n \Rightarrow b_n) \quad = \quad 1+\max\{r(a_1), r(b_1), \ldots, r(a_n), r(b_n)\}$$

Thus in contrast to the old rank function r, the new rank function r' adds the cardinality n of the function graph. In this way, the rank becomes a more precise measure of the size of a value.

To complete these considerations, we must ensure that the semantics presented in Section 4.4 remains well-formed when switching to the new metric space of ideals based on the rank function r'. One way to prove this is to adapt the proofs in the original work on the ideal model of types [MPS86]. This causes no problems. However, we present here a different and simpler proof. The important observation is expressed in the following lemma.

Lemma 4.1 *For all ideals $I_1, I_2 \in \mathcal{J}_0(\mathbf{V})$, $d_{r'}(I_1, I_2) \leq d_r(I_1, I_2)$.*

Proof [Sketch]: For the two rank functions r and r' we have $r(v) \leq r'(v)$ for all finite values $v \in \mathbf{V}^\circ$. Let $I_1, I_2 \in \mathcal{J}_0(\mathbf{V}^\circ)$ be any order ideals. Assume I_1, I_2 are different. From the above it follows

$$\min\{r(v) \mid v \in I_1 \ominus I_2\} \leq \min\{r'(v) \mid v \in I_1 \ominus I_2\}$$

i.e. $c_r(I_1, I_2) \leq c_{r'}(I_1, I_2)$. Consequently, $d_{r'}(I_1, I_2) \leq d_r(I_1, I_2)$. □

From this lemma we may derive:

Lemma 4.2 *The identity mapping $I \mapsto I$ is continuous as a mapping between $(\mathcal{J}(\mathbf{V}), d_r)$ and $(\mathcal{J}(\mathbf{V}), d_{r'})$.*

Since continuous mappings preserve limits, we have that the semantic definition $\mathcal{D}$ is well-defined in the modified metric space $(\mathcal{J}(\mathbf{V}), d_{r'})$.

5 Encoding of Types

The main idea of the approach presented here is to represent the structure of a value in such a way that the set of trees corresponding to a type is regular. Since the inclusion problem for regular sets of trees is decidable, we may in this way gain a decision procedure for the type inclusion problem.

We present succinct requirements to the encoding of types which are sufficient to ensure respectively soundness and completeness. We show that the soundness requirements is fulfilled for the type language presented in Section 4 and the encoding presented in [Dam94a]. As a byproduct we have a new soundness proof which is free of the rather adverse assumption that all types must be infinite.

5.1 Representation of Values

To motivate the presentation of values by trees, consider a tuple $<x, y>$. If t_x and t_y are trees representing respectively x and y, then we may represent the tuple $<x, y>$ by a binary tree $p(t_x, t_y)$. For function values it becomes slightly more difficult because a function does not induce a tree like structure in the same simpler manner as a product. One solution is to represent the graph $grf_{std}(f)$ of a function f:

$$grf_{std}(f) \quad = \quad \{<x,y> \in \mathbf{V} \times \mathbf{V} \mid y = f(x)\}$$

However, since step functions may have an infinite domain, such graphs may be infinite even for finite values. In order to obtain a finite representation, we introduce the notion of *minimalized graphs*. Consider a function f and two arguments x_1 and x_2 such that x_1 is strictly less defined than x_2, i.e. $x_1 \sqsubset x_2$. From the monotonicity of f we have that $f(x_1) \sqsubseteq f(x_2)$. If $f(x_1) = f(x_2)$ then the pair $<x_2, f(x_2)>$ does not really add any new information. To obtain a minimalized graph from a "standard" graph, we remove such pairs. More formally, the minimalized graph $grf(f)$ of a function f is defined by:

$$grf(f) \quad = \quad \{<x,y> \in \mathbf{V} \times \mathbf{V} \mid y = f(x) \wedge \forall x' \sqsubset x \,.\, f(x') \sqsubset y\}$$

It is not difficult to see that the set

$$\{a \Rightarrow b \mid <a,b> \in grf(f)\}$$

corresponds to the minimal set of step functions corresponding to f. Since step functions have finite codomains, the minimalized graph of a finite function value is a finite set. In the sequel we will assume that all graphs are minimalized.

A graph $grf(f)$ may be encoded as a sequence of pairs whose set of elements is exactly $grf(f)$. A sequence $<t_1, t_2, \ldots, t_n>$ is represented by a right recursive tree

$$g(t_1, g(t_2, \ldots g(t_n, nil) \ldots))$$

where g is a new binary symbol and *nil* is a new 0-ary symbol. By *seqs* we denote the set of all such sequences. The function *elems* yields the set of elements of such a sequence. We make use of the following abbreviations for expressions denoting sets of such sequences.

$$\begin{array}{rcl}[\zeta_1, \zeta_2, \ldots, \zeta_n] & = & g(\zeta_1, g(\zeta_2, \ldots g(\zeta_n, \mathit{nil}) \ldots)) \\ [\zeta]^+ & = & \mathsf{fix}\, t.g(\zeta, t) + g(\zeta, \mathit{nil}) \text{ where } t \text{ is not free in } \zeta\end{array}$$

Since the same function graph may be represented by many different sequences, the representation is lifted to sets of values. For each basic domain $\mathbf{B}_i$, we introduce a new 0-ary symbol κ_i. A mapping $\mathcal{T}$ from finite values to trees over the alphabet $\Sigma = \{\kappa_1, \ldots, \kappa_n, p, g, \mathit{nil}\}$ may now be defined.

$$\begin{array}{rcl}\mathcal{T}(\bot) & = & \emptyset \\ \mathcal{T}(b_{ij} : \mathbf{B}_i) & = & \{\kappa_i\} \\ \mathcal{T}(<x, y> : (\mathbf{V} \times \mathbf{V})^\circ) & = & \{p(t_1, t_2) \,|\, t_1 \in \mathcal{T}(x) \wedge t_2 \in \mathcal{T}(y)\} \\ \mathcal{T}(f : (\mathbf{V} \to \mathbf{V})^\circ) & = & \{t \in \mathit{seqs} \,|\, \forall t_p \in \mathit{elems}(t)\,. \\ & & \qquad \exists <x, y> \in \mathit{grf}(f)\,. \\ & & \qquad\quad t_p \in \mathcal{T}(<x, y>) \wedge \\ & & \quad \forall <x, y> \in \mathit{grf}(f)\,. \\ & & \qquad \exists t_p \in \mathit{elems}(t)\,. \\ & & \qquad\quad t_p \in \mathcal{T}(<x, y>)\}\end{array}$$

The function $\mathcal{T}$ is lifted to also work on sets of values as follows:

$$\mathcal{T}(I : \mathcal{P}(\mathbf{V}^\circ)) \quad = \quad \bigcup_{v \in I} \mathcal{T}(v)$$

It is a trivial consequence of this definition that $\mathcal{T}$ is monotonic with respect to inclusion. As an approximation of the representation of all finite values, we define

$$\zeta_{\mathsf{T}} = \mathsf{fix}\, t.\kappa_1 + \ldots + \kappa_n + p(t, t) + [p(t, t)]^+$$

Next we consider a function $\mathcal{V}$ going in the opposite direction of $\mathcal{T}$, i.e. a function mapping trees into values the represented values. $\mathcal{V}$ is defined in terms of $\mathcal{T}$.

$$\mathcal{V}(t) \quad = \quad \{v \in \mathbf{V}^\circ \,|\, t \in \mathcal{T}(v)\}$$

It is immediate from this definition that $v \in \mathcal{V}(t)$ iff $t \in \mathcal{T}(v)$. The function is lifted to work on sets of trees in a similar way as for $\mathcal{T}$. The bottom value $\bot$ is included by default.

$$\begin{array}{rcl}\mathcal{V}(F) & = & \{\bot\} \cup \bigcup_{t \in F} \mathcal{V}(t) \\ & = & \{\bot\} \cup \{v \in \mathbf{V}^\circ \,|\, \exists t \,.\, t \in F \wedge t \in \mathcal{T}(v)\}\end{array} \tag{4}$$

Like $\mathcal{T}$, $\mathcal{V}$ is monotonic with respect to inclusion. With the definition of $\mathcal{V}$, the representation of function values may be expressed in a more comprehensive way.

$$\begin{array}{rcl}\mathcal{T}(f : (\mathbf{V} \to \mathbf{V})^\circ) & = & \{t \in \mathit{seqs} \,|\, \mathit{elems}(t) \subseteq \mathcal{T}(\mathit{grf}(f)) \wedge \\ & & \qquad \mathit{grf}(f) \subseteq \mathcal{V}(\mathit{elems}(t))\}\end{array}$$

5.2 Representation of Types

We are aiming at an encoding $\mathcal{R}$ of types into regular tree expressions such that the diagram

$$\begin{tikzcd}
\mathbf{TExp} \arrow[r, "\mathcal{D}"] \arrow[d, "\mathcal{R}"'] & \mathcal{J}(\mathbf{V}) \arrow[r, leftrightarrow, "I \mapsto I^\circ"] & \mathcal{J}_0(\mathbf{V}^\circ) \arrow[r, "I \mapsto I"] & \mathcal{P}(\mathbf{V}^\circ) \\
\mathbf{RE}(\Sigma, \mathbf{Var}) \arrow[rrr, "\mathcal{I}"'] & & & H_\Sigma \arrow[u, "\mathcal{V}"']
\end{tikzcd} \tag{5}$$

commutes for all types and suitable type and forest environments. By the monotonicity of $\mathcal{V}$ with respect to inclusion and the order isomorphism of $\mathcal{J}(\mathbf{V})$ and $\mathcal{J}_0(\mathbf{V}^\circ)$, it then follows that $\mathcal{D}[\![\tau_1]\!]\rho \subseteq \mathcal{D}[\![\tau_2]\!]\rho$ whenever $\mathcal{I}[\![\mathcal{R}[\![\tau_1]\!]]\!]\eta \subseteq \mathcal{I}[\![\mathcal{R}[\![\tau_2]\!]]\!]\eta$ (for suitable ρ and η).

The bottom type $\perp$, type variables plus union, intersection and recursive types are encoded straightforwardly as the corresponding operations on regular tree expressions.

$$\begin{aligned}
\mathcal{R}[\![\perp]\!] &= \mathsf{bot} \\
\mathcal{R}[\![t]\!] &= t \\
\mathcal{R}[\![\tau_1 \sqcup \tau_2]\!] &= \mathcal{R}[\![\tau_1]\!] + \mathcal{R}[\![\tau_2]\!] \\
\mathcal{R}[\![\tau_1 \sqcap \tau_2]\!] &= \mathcal{R}[\![\tau_1]\!] * \mathcal{R}[\![\tau_2]\!] \\
\mathcal{R}[\![\mu t.\tau]\!] &= \mathsf{fix}\, t.\mathcal{R}[\![\tau]\!]
\end{aligned}$$

For the basic type constructors, we will allow some freedom. We give three succinct soundness requirements. They capture the properties which are essential in order to make the Diagram (5) commute.

The first requirement says that each all values—except $\perp$ which is a member of all types—must have a tree representation.

Soundness Requirement 1 $\forall v \in \mathbf{V}^\circ \setminus \{\perp\} \,.\, \mathcal{T}(v) \neq \emptyset$

To formalize the next requirement, we introduce two predicates P_a and P_b.

$$\begin{aligned}
P_a(\tau) &= \mathcal{T}(\mathcal{D}^\circ[\![\tau]\!]\rho) \subseteq \mathcal{I}[\![\mathcal{R}[\![\tau]\!]]\!]\eta \\
P_b(\tau) &= \mathcal{V}(\mathcal{I}[\![\mathcal{R}[\![\tau]\!]]\!]\eta) \subseteq \mathcal{D}^\circ[\![\tau]\!]\rho
\end{aligned}$$

The environments ρ and η are assumed defined in the context. For P_a the intuition is that the encoding of a type τ should contain all trees representing values of type τ. In other words, P_a accepts a type τ if the encoding is *representation closed.* The predicate P_b accepts a type τ if its encoding does not contain trees representing values outside type τ. Putting it in another way, P_b accepts a type τ, if the type holds all values whose structures are represented by a tree in $\mathcal{R}[\![\tau]\!]$. That is, τ is *closed under structural equivalence.*

Soundness Requirement 2 *For all basic type constructors c, type environments ρ and forest environments η it is true that* *(a)* $P_a(\tau_1) \wedge \ldots \wedge P_a(\tau_n) \Rightarrow P_a(c(\tau_1, \ldots, \tau_n))$ *and*

(b) $P_b(\tau_1) \wedge \ldots \wedge P_b(\tau_n) \Rightarrow P_b(c(\tau_1, \ldots, \tau_n))$.

where n is the arity of c.

Notice that the requirements specify necessary relations between $\mathcal{T}$ and $\mathcal{R}$ rather than a requirements on $\mathcal{R}$ solely. Furthermore, $\mathcal{R}[\![\tau]\!]$ may contain trees which are not in $\mathcal{T}(\mathcal{D}[\![\tau]\!]\rho)$. However, such trees should be meaningless in the sense that they do not represent anything. This freedom is important as it may not be possible to define $\mathcal{T}$ such that $\mathcal{T}(\mathcal{D}[\![\tau]\!]\rho)$ becomes regular.

Lemma 5.1 *Assume Soundness Requirement 1 is satisfied. For all types τ, if $P_a(\tau)$ and $P_b(\tau)$ are true then*

$$\mathcal{D}^\circ[\![\tau]\!]\rho \quad = \quad \mathcal{V}(\mathcal{I}[\![\mathcal{R}[\![\tau]\!]]\!]\eta)$$

Proof: From $P_a(\tau)$ we get

$$\begin{aligned}
\mathcal{V}(\mathcal{I}[\![\mathcal{R}[\![\tau]\!]]\!]\eta) &\supseteq \mathcal{V}(\mathcal{T}(\mathcal{D}^\circ[\![\tau]\!]\rho)) \\
&= \{v \in \mathbf{V}^\circ \mid \exists t \,.\, t \in \mathcal{T}(\mathcal{D}^\circ[\![\tau]\!]\rho) \wedge t \in \mathcal{T}(v)\} \cup \{\bot\} \\
&= \{v \in \mathbf{V}^\circ \mid \exists t \,.\, \mathcal{V}(t) \cap \mathcal{D}^\circ[\![\tau]\!]\rho \neq \emptyset \wedge t \in \mathcal{T}(v)\} \cup \{\bot\}
\end{aligned}$$

Assume $v \in \mathcal{D}^\circ[\![\tau]\!]\rho \setminus \{\bot\}$. Then for all $t \in \mathcal{T}(v)$, $\mathcal{V}(t) \cap \mathcal{D}^\circ[\![\tau]\!]\rho \neq \emptyset$. Since $\mathcal{T}(v) \neq \emptyset$ (Soundness Requirement 1) it follows that $\exists t \in \mathcal{T}(v) \,.\, \mathcal{V}(t) \cap \mathcal{D}^\circ[\![\tau]\!]\rho \neq \emptyset$, and consequently $v \in \mathcal{V}(\mathcal{T}(\mathcal{D}^\circ[\![\tau]\!]\rho))$. This proves that $\mathcal{D}^\circ[\![\tau]\!]\rho \subseteq \mathcal{V}(\mathcal{T}(\mathcal{D}^\circ[\![\tau]\!]\rho))$ and completes the proof. $\Box$

At this stage it may seem a little strange that the requirement expressed by the predicate P_b, is an inclusion when P_a and P_b together express a requirement $\mathcal{D}^\circ[\![\tau]\!]\rho = \mathcal{V}(\mathcal{I}[\![\mathcal{R}[\![\tau]\!]]\!]\eta)$ which is stronger than P_b. The motivation for doing so is that we think the inclusion is more intuitive and therefore serves as a better basis for defining encoding for actual type constructors.

Under the assumption that $\mathcal{T}$ and $\mathcal{R}$ meet Soundness Requirement 2, it may be proved that the properties expressed by the predicates P_a and P_b are fulfilled for all non-recursive types. In order to deal with type variables, we introduce the notion of semi-agreement which expresses that P_a and P_b are fulfilled for all type variables.

Definition 5.1 (Semi-agreement) *A type environment (not necessarily proper) ρ and a forest environment η semi-agree if for all variables $t \in$ **Var**:*

(a) $\mathcal{T}(\rho(t)^\circ) \subseteq \eta(t)$, *and*

(b) $\mathcal{V}(\eta(t)) \subseteq \rho(t)^\circ$

Lemma 5.2 *Let ρ be any (possibly improper) type environment semi-agreeing with a forest environment η. Assume that all basic type constructors satisfy Soundness Requirement 2. Then for all types τ not containing recursion:*

(a) $\mathcal{T}(\mathcal{D}[\![\tau]\!]\rho) \subseteq \mathcal{I}[\![\mathcal{R}[\![\tau]\!]]\!]\eta$, *and*

(b) $\mathcal{V}(\mathcal{I}[\![\mathcal{R}[\![\tau]\!]]\!]\eta) \subseteq \mathcal{D}[\![\tau]\!]\rho$

Proof: The proof is by the induction on the structure of τ.

case $\bot$**.** Easy.

case t**.** Follows directly from Definition 5.1.

case $\tau_1 \sqcup \tau_2$**.** Easy.

case $\tau_1 \sqcap \tau_2$**.** (a) From the definition of $\mathcal{D}$ and the component-wise definition of $\mathcal{T}$ it is immediate that

$$\begin{aligned}\mathcal{T}(\mathcal{D}^{\circ}[\![\tau_1 \sqcap \tau_2]\!]\rho) &= \mathcal{T}(\mathcal{D}^{\circ}[\![\tau_1]\!]\rho \cap \mathcal{D}^{\circ}[\![\tau_2]\!]\rho) \\ &\subseteq \mathcal{T}(\mathcal{D}^{\circ}[\![\tau_1]\!]\rho) \cap \mathcal{T}(\mathcal{D}^{\circ}[\![\tau_2]\!]\rho)\end{aligned}$$

It remains to proved that there does not exist any value v in the symmetric difference of $\mathcal{D}^{\circ}[\![\tau_1]\!]\rho$ and $\mathcal{D}^{\circ}[\![\tau_2]\!]\rho$ such that $\mathcal{T}(v)$ intersects with $\mathcal{T}(\mathcal{D}^{\circ}[\![\tau_1]\!]\rho) \cap \mathcal{T}(\mathcal{D}^{\circ}[\![\tau_2]\!]\rho)$. Assume there exists such a value v, say $v \in \mathcal{D}^{\circ}[\![\tau_1]\!]\rho$, and let $t \in \mathcal{T}(v) \cap \mathcal{T}(\mathcal{D}^{\circ}[\![\tau_1]\!]\rho) \cap \mathcal{T}(\mathcal{D}^{\circ}[\![\tau_2]\!]\rho)$. By the induction hypothesis we have $t \in \mathcal{I}[\![\mathcal{R}[\![\tau_2]\!]]\!]\eta$. But since $\mathcal{V}(\mathcal{I}[\![\mathcal{R}[\![\tau_2]\!]]\!]\eta) \subseteq \mathcal{D}[\![\tau_2]\!]\rho$ we have $v \in \mathcal{D}^{\circ}[\![\tau_2]\!]\rho$ which contradicts the assumption.

(b) Analogous to (a).

case $c(\tau_1, \ldots, \tau_n)$ where c is a basic type constructor of arity n. The proof follows from the assumptions.

From Lemma 5.1 and 5.2, it follows immediately that Diagram (5) commutes for all types not containing recursion.

Lemma 5.3 *Let τ be any type not containing recursion, ρ any (possibly improper) type environment, and η any forest environment semi-agreeing with ρ. Assume Soundness Requirement 1 and 2 are satisfied. Then*

$$\mathcal{D}^{\circ}[\![\tau]\!]\rho \;=\; \mathcal{V}(\mathcal{I}[\![\mathcal{R}[\![\tau]\!]]\!]\eta)$$

We do, however, still need to consider recursion. Before doing so, we consider the encoding of the basic constructors of the sample type language. The encoding presented in [Dam94a] is:

$$\begin{aligned}\mathcal{R}[\![\iota_i]\!] &= \kappa_i \\ \mathcal{R}[\![\tau_1 \times \tau_2]\!] &= p(\mathcal{R}[\![\tau_1]\!], \mathcal{R}[\![\tau_2]\!]) \\ \mathcal{R}[\![\tau_1 \rightarrow \tau_2]\!] &= [p(\mathcal{R}[\![\tau_1]\!], \mathcal{R}[\![\tau_2]\!]) + p(\overline{\mathcal{R}[\![\tau_1]\!]} * \zeta_{\top}, \zeta_{\top})]^{+}\end{aligned}$$

At first sight the encoding of function types may seem a little strange. As described above, functions are encoded as the sequences (representing graphs) whose elements (pairs) represent function graph elements. Now reconsider the semantics of function types. Whenever the first component of a function graph element is of type τ_1 then the second component must be of type τ_2, i.e. values of type τ_1 are mapped into values of type τ_2. However, if the first component of a function graph element is not of type τ_1 then there is no restriction on the second component. These two possibilities are expressed by the two alternatives $p(\mathcal{R}[\![\tau_1]\!], \mathcal{R}[\![\tau_2]\!])$ and $p(\overline{\mathcal{R}[\![\tau_1]\!]} * \zeta_\top, \zeta_\top)$ for sequence elements.

Lemma 5.4 *For the sample language introduced in Section 4, the tree representation $\mathcal{T}$ and the encoding $\mathcal{R}$ fulfills Soundness Requirement 1 and 2.*

The proof may be found in Appendix A. It makes use of the following general lemma which is also proved in Appendix A.

Lemma 5.5 *For all types τ (and environments ρ and η), if $P_a(\tau)$ and $P_b(\tau)$ then*

(i) $\mathcal{V}(\mathcal{I}[\![\overline{\mathcal{R}[\![\tau]\!]}]\!]\eta) = \overline{\mathcal{V}(\mathcal{I}[\![\mathcal{R}[\![\tau]\!]]\!]\eta)} \cup \{\bot\}$

(ii) $\overline{\mathcal{T}(\mathcal{D}^\circ[\![\tau]\!]\rho)} = \mathcal{T}(\overline{\mathcal{D}^\circ[\![\tau]\!]\rho}) \cap \mathcal{T}(\mathbf{V}^\circ)$

In order to generalize Lemma 5.3 to recursive types, we would like $\mathcal{V}$ to be a continuous mapping between the metric space of sets of values and the metric space of forests. In terms of closeness, this means that for all forests F_0 and $m > 0$ it must be possible to come up with an n such that for all F, $c_{hg}(F_0, F) \leq n$ whenever $c_r(\mathcal{V}(F_0), \mathcal{V}(F)) \leq m$. Since some values may be represented by arbitrary big trees, we must first of all restrict our attention to forests which contain a least representation. This requirement is most easily accommodated by requiring all forests to be representation closed.

Unfortunately, this is not always sufficient to ensure continuity of $\mathcal{V}$. As an example of this problem, the encoding of our sample type language leads to a $\mathcal{V}$ which is not continuous—even not for representation closed forests. Basically, the problem is that from the rank of a value we cannot determine the height of the least tree representing that value. For the type language in Section 4, it is the function values which cause the problem: the rank of a function value does not say anything about the cardinality of its graph. We need a closer connection between the rank a value and the height the least corresponding tree. The important property is that the distance between two forests does not "jump" (or the closeness does not "shrink") when applying $\mathcal{V}$. This is the essence of the third soundness requirement.

Soundness Requirement 3 *There must exist a rank function r on $\mathbf{V}^\circ$ such that $\mathcal{D}$ is well-defined, and there is constant c such that for all values $v \in \mathbf{V}^\circ$,*

$$\min\{hg(t) \mid t \in \mathcal{T}(v)\} \quad \leq \quad c\, r(v)$$

It is not difficult to see that the alternative rank function r' introduced in Section 4.5 satisfies this requirement.

Lemma 5.6 *If the rank r for finite values satisfies Soundness Requirement 3 then the mapping $\mathcal{V}$ is a continuous mapping from the representation closed part of $(\mathcal{P}(H_\Sigma), d_{hg})$ to $(\mathcal{P}(\mathbf{V}^\circ), d_r)$.*

Proof: It is sufficient to prove that for all $F_0, F \in \mathcal{P}(H_\Sigma)$ and all $m > 0$, if $c_r(\mathcal{V}(F_0), \mathcal{V}(F)) \leq m$ then $c_{hg}(F_0, F) \leq m$. Assume $c_r(\mathcal{V}(F_0), \mathcal{V}(F)) \leq m$. Let $v_w \in \mathcal{V}(F_0) \ominus \mathcal{V}(F)$ be such that $v_w \leq m$. By the element-wise definition of $\mathcal{V}$, the set $\{t \mid v_w \in \mathcal{V}(t)\} \cap (F_0 \cup F)$ is a non-empty set of witnesses of F_0 and F. Since F_0 and F are representation closed, this set is equal to $\{t \mid v_w \in \mathcal{V}(t)\}$. By Soundness Requirement 3 is follows that $c_{hg}(F_0, F) \leq m$. □

Lemma 5.7 *Let τ be type, ρ any* proper *type environment and η any forest environment semi-agreeing with ρ. Assume Soundness Requirement 1, 2 and 3 are satisfied. Then it holds that:*

$$\mathcal{D}^\circ[\![\tau]\!]\rho \quad = \quad \mathcal{V}(\mathcal{I}[\![\mathcal{R}[\![\tau]\!]]\!]\eta)$$

Proof: For every recursive type $\mu t.\tau'$, we have that $\mathcal{D}^\circ[\![\mu t.\tau']\!]\rho$ is the limit of the Cauchy sequence $<D^n(\{\perp\})>_{n\geq 0}$ where $D \equiv \lambda I\,.\,\mathcal{D}^\circ[\![\tau']\!]\rho\{I/x\}$. Also, for all $i > 0$, $D^i(\mathcal{D}^\circ[\![\tau'']\!]\rho) = D^{i-1}(\mathcal{D}^\circ[\![\tau'[\tau''/t]]\!]\rho)$. Therefore, the denotation of τ may be described as the limit of a Cauchy sequence of types which do not contain $\mu t.\tau$ as a subexpression. By simultaneous unfolding all recursive subexpressions of τ, we can obtain a Cauchy sequence of types with the denotation of τ as the limit but without types containing recursion. Since $\mathcal{V}(\mathcal{I}[\![\mathcal{R}[\![\tau']\!]]\!]\eta) = \mathcal{D}^\circ[\![\tau']\!]\rho$ for all non-recursive types τ', and since V is continuous, it follows that $\mathcal{V}(\mathcal{I}[\![\mathcal{R}[\![\tau]\!]]\!]\eta) = \mathcal{D}^\circ[\![\tau]\!]\rho$ for types τ containing recursion. □

Theorem 5.1 (Soundness of subtyping) *Let $\mathcal{R}$ be an encoding fulfilling Soundness Requirement 1, 2 and 3. Let τ_1, τ_2 be any types. If for all forests environments η, $\mathcal{I}[\![\mathcal{R}[\![\tau_1]\!]]\!]\eta \subseteq \mathcal{I}[\![\mathcal{R}[\![\tau_2]\!]]\!]\eta$ then there exists a (semi-agreeable) type environment ρ such that $\mathcal{D}[\![\tau_1]\!]\rho \subseteq \mathcal{D}[\![\tau_2]\!]\rho$.*

We now shortly sketch how to obtain completeness. The proofs may be adapted from similar proofs in [Dam94a]. First, we define the predicate P_c.

$$P_c(\tau) = \forall t \in \mathcal{I}[\![\mathcal{R}[\![\tau]\!]]\!]\eta\,.\,\mathcal{V}(t) \neq \emptyset$$

The predicate P_c essentially requires that the encoding of a type τ may not contain "garbage", i.e. trees not representing any value.

Completeness Requirement 1 *For all basic type constructors c, type environments ρ and forest environments η it is true that*

$$P_c(\tau_1) \wedge \ldots \wedge P_c(\tau_n) \Rightarrow P_c(c(\tau_1, \ldots, \tau_n))$$

where n is the arity of c.

Completeness Requirement 2 *There must exist a rank function r on $\mathbf{V}^\circ$ such that $\mathcal{D}$ is well-defined, and there is constant c such that for all values $v \in \mathbf{V}^\circ$,*

$$c \min\{hg(t) \mid t \in \mathcal{T}(v)\} \quad \geq \quad r(v)$$

The latter requirement is sufficient to ensure that $\mathcal{V}$ is continuous as a mapping from $(\mathcal{P}(\mathbf{V}^\circ), d_r)$ to $(\mathcal{P}(\mathbf{V}^\circ), d_{hg})$. Both requirements are satisfied for our sample language provided that all non-empty types are assumed infinite. To make type variables meet Completeness Requirement 1, the notion of agreement is necessary.

Definition 5.2 (Agreement) *A type environments ρ and a forest environment η agree if for all variables $t \in \mathbf{Var}$:*

(a) $\mathcal{T}(\rho(t)^\circ) = \eta(t)$

(b) $\mathcal{V}(\eta(t)) \subseteq \rho(t)^\circ$

Lemma 5.8 *Let τ be type, ρ any* proper *type environment and η any forest environment agreeing with ρ. Assume that both Soundness Requirement 1, 2 and 3, and Completeness Requirement 1 and 2 are satisfied.*

$$\mathcal{T}(\mathcal{D}^\circ[\![\tau]\!]\rho) \quad = \quad \mathcal{I}[\![\mathcal{R}[\![\tau]\!]]\!]\eta$$

Theorem 5.2 (Completeness of subtyping) *Let $\mathcal{R}$ be an encoding fulfilling Soundness Requirement 1, 2 and 3, and Completeness Requirement 1 and 2. Let τ_1, τ_2 be any types. If $\mathcal{D}[\![\tau_1]\!]\rho \subseteq \mathcal{D}[\![\tau_2]\!]\rho$ for all (agreeable) type environments, then there exists a forest environment η such that $\mathcal{I}[\![\mathcal{R}[\![\tau_1]\!]]\!]\eta \subseteq \mathcal{I}[\![\mathcal{R}[\![\tau_2]\!]]\!]\eta$.*

6 Other Examples

We shall here shortly sketch how some variations on the semantics of our sample language may be reflected in the encoding. First, assume we want to distinguish $\perp$ and $\lambda x\,.\,\perp$. This just means that a function graph may be empty. The encoding may account for this by allowing empty sequences in the representation of function types. This also rules out the inclusion $\top \rightarrow \perp \leq \perp$.

Next, assume that types are interpreted in a domain including non-strict functions and products. This may be accommodated by the encodings (*undef* is a new 0-ary symbol):

$$\begin{array}{rcl}
\mathcal{R}[\![\tau_1 \times \tau_2]\!] & = & p(\mathit{undef} + \mathcal{R}[\![\tau_1]\!], \mathit{undef} + \mathcal{R}[\![\tau_2]\!]) \\
\mathcal{R}[\![\tau_1 \rightarrow \tau_2]\!] & = & [p(\mathit{undef} + \mathcal{R}[\![\tau_1]\!], \mathcal{R}[\![\tau_2]\!]) + p(\overline{\mathcal{R}[\![\tau_1]\!]} * \zeta_\top, \zeta_\top)]^* \text{ where} \\
\zeta_\top & = & \mathsf{fix}\, t.\kappa_1 + \ldots + \kappa_n + p(\mathit{undef} + t, \mathit{undef} + t) + [p(\mathit{undef} + t, t)]^* \\
[\zeta]^* & = & \mathsf{fix}\, t.g(\zeta, t) + \mathit{nil} \text{ where } t \text{ is not free in } \zeta
\end{array}$$

7 Concluding Remarks

We have presented a generic technique on encoding of types into regular tree expressions. The approach has been applied to a sample language which constitute the core of many languages. The complexity results for set constraints are quite discouraging ([NEXPTIME]) [AKVW93], but it is not clear what these worst case results mean in practice for our subtyping approach.

References

[AC91] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages (POPL'91)*, pages 104–118, ACM Press, 1991.

[AKVW93] Alexander Aiken, Dexter Kozen, Moshe Vardi, and Edward L. Wimmers. *The Complexity of Set Constraints.* Technical Report 93-1352, Computer Science Department, Cornell University, 1993.

[AM91] Alexander Aiken and Brian M. Murphy. Implementing regular tree expressions. In J. Hughes, editor, *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture (FPCA'91), Cambridge, MA, USA, August 1991,* volume 523 of *Lecture Notes in Computer Science*, pages 427–447, Springer-Verlag, 1991.

[AW92] Alexander Aiken and Edward L. Wimmers. Solving systems of set constraints (extended abstract). In *Proceedings of the Seventh Annual IEEE Symposium on Logic in Computer Science (LICS'92), Santa Cruz, California, June 1992*, pages 329–340, IEEE Computer Society Press, 1992.

[AW93] Alexander Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In *Proceedings of the 6th ACM Conference on Functional Programming Languages and Computer Architecture (FPCA'93), Copenhagen, Denmark, June 1993*, pages 31–41, ACM Press, 1993.

[CF91] Robert Cartwright and Mike Fagan. Soft typing. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI'91)*, pages 278–292, ACM Press, 1991.

[Dam94a] Flemming M. Damm. Subtyping with union types, intersection types and recursive types. In *Proceedings of the International Symposium on Theoretical Aspects of Computer Software (TACS'94), Sendai, Japan, April 1994,* volume 789 of *Lecture Notes in Computer Science*, pages 687–706, Springer-Verlag, 1994.

[Dam94b] Flemming M. Damm. *Type Inference with Union Types, Intersection Types and Recursive Types.* Technical Report, INRIA, Institut National de Recherche en Informatique et en Automatique, 1994. In preparation.

[FP91] Tim Freeman and Frank Pfenning. Refinement types for ML. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI'91)*, pages 268–277, ACM Press, 1991.

[GS84] Ferenc Gécseg and Magnus Steinby. *Tree Automata.* Akadémiai Kiadó, Budapest, 1984.

[GTT93] R. Gilleron, S. Tison, and M. Tommasi. Solving systems of set constraints with negated subset relationships. In *Proceedings of the 34th Annuel Symposium on Foundations of Computer Science (FOCS'93)*, pages 372–380, IEEE Computer Society Press, 1993.

[Hin92] J. Roger Hindley. Types with intersection: an introduction. *Formal Aspect of Computing*, 4(5):470–486, 1992.

[MPS86] David MacQueen, Gordon Plotkin, and Ravi Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71:95–130, 1986.

[MR85] Prateek Mishra and Uday S. Reddy. Declaration-free type checking. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages (POPL'85)*, pages 7–21, 1985.

A Proofs

Proof of Lemma 5.5:

(i) Notice that $\mathcal{I}[\![\overline{\mathcal{R}[\![\tau]\!]}]\!]\eta = \overline{\mathcal{I}[\![\mathcal{R}[\![\tau]\!]]\!]\eta}$. Using Equation (4) we may derive:

$$\begin{array}{rcl} \mathcal{V}(\overline{\mathcal{I}[\![\mathcal{R}[\![\tau]\!]]\!]\eta}) & = & \{v \in \mathbf{V}^\circ \mid \mathcal{T}(v) \cap \overline{\mathcal{I}[\![\mathcal{R}[\![\tau]\!]]\!]\eta} \neq \emptyset\} \\ \overline{\mathcal{V}(\mathcal{I}[\![\mathcal{R}[\![\tau]\!]]\!]\eta)} & = & \{v \in \mathbf{V}^\circ \mid \mathcal{T}(v) \subseteq \overline{\mathcal{I}[\![\mathcal{R}[\![\tau]\!]]\!]\eta}\} \setminus \{\bot\} \end{array}$$

Since $A \cap B = \emptyset$ iff $A \subseteq \overline{B}$, it just remains to prove that for all $v \neq \bot$

$$\mathcal{T}(v) \subseteq \mathcal{I}[\![\mathcal{R}[\![\tau]\!]]\!]\eta \quad \Leftrightarrow \quad \mathcal{T}(v) \cap \mathcal{I}[\![\mathcal{R}[\![\tau]\!]]\!]\eta \neq \emptyset$$

($\Rightarrow$) Follows directly from Soundness Requirement 1.

($\Leftarrow$) Rewriting $P_b(\tau)$ we derive $\forall t \,.\, t \in \mathcal{I}[\![\mathcal{R}[\![\tau]\!]]\!]\eta \Rightarrow \forall v \,.\, t \in \mathcal{T}(v) \Rightarrow v \in \mathcal{D}^\circ[\![\tau]\!]\rho$ which in turn is equivalent to

$$\forall v \,.\, \forall t \,.\, t \in \mathcal{I}[\![\mathcal{R}[\![\tau]\!]]\!]\eta \Rightarrow t \in \mathcal{T}(v) \Rightarrow v \in \mathcal{D}^\circ[\![\tau]\!]\rho$$

This proves that $v \in \mathcal{D}^\circ[\![\tau]\!]\rho$. From $P_a(\tau)$ it follows that $\mathcal{T}(v) \subseteq \mathcal{I}[\![\mathcal{R}[\![\tau]\!]]\!]\eta$.

(ii) Combining $P_a(\tau)$ and $P_b(\tau)$ we get $\mathcal{V}(\mathcal{T}(\mathcal{D}^\circ[\![\tau]\!]\rho)) \subseteq \mathcal{D}^\circ[\![\tau]\!]\rho$.

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathcal{V}(\mathcal{T}(\mathcal{D}^\circ[\![\tau]\!]\rho)) \subseteq \mathcal{D}^\circ[\![\tau]\!]\rho} \\
\Leftrightarrow & \forall t \in \mathcal{T}(\mathcal{D}^\circ[\![\tau]\!]\rho) \,.\, \mathcal{V}(t) \subseteq \mathcal{D}^\circ[\![\tau]\!]\rho \\
\Leftrightarrow & \forall t \,.\, \mathcal{V}(t) \cap \mathcal{D}^\circ[\![\tau]\!]\rho \neq \emptyset \Rightarrow \mathcal{V}(t) \subseteq \mathcal{D}^\circ[\![\tau]\!]\rho \\
\Leftrightarrow & \forall t \,.\, \mathcal{V}(t) \not\subseteq \mathcal{D}^\circ[\![\tau]\!]\rho \Rightarrow \mathcal{V}(t) \cap \mathcal{D}^\circ[\![\tau]\!]\rho = \emptyset \\
\Leftrightarrow & \forall t \,.\, \mathcal{V}(t) \cap \overline{\mathcal{D}^\circ[\![\tau]\!]\rho} \neq \emptyset \Rightarrow \mathcal{V}(t) \cap \mathcal{D}^\circ[\![\tau]\!]\rho = \emptyset \\
\Leftrightarrow & \{t \mid \mathcal{V}(t) \cap \overline{\mathcal{D}^\circ[\![\tau]\!]\rho} \neq \emptyset\} \subseteq \{t \mid \mathcal{V}(t) \cap \mathcal{D}^\circ[\![\tau]\!]\rho = \emptyset\} \\
\Leftrightarrow & \mathcal{T}(\overline{\mathcal{D}^\circ[\![\tau]\!]\rho}) \subseteq \overline{\mathcal{T}(\mathcal{D}^\circ[\![\tau]\!]\rho)}
\end{array}$$

Next we have

$$\begin{array}{rcl}
\overline{\mathcal{T}(\mathcal{D}^\circ[\![\tau]\!]\rho)} & = & \{t \mid \mathcal{V}(t) \cap \mathcal{D}^\circ[\![\tau]\!]\rho = \emptyset\} \\
& = & \{t \mid \mathcal{V}(t) \subseteq \overline{\mathcal{D}^\circ[\![\tau]\!]\rho}\} \\
& \subseteq & \overline{\mathcal{T}(\mathbf{V}^\circ)} \cup \{t \mid \mathcal{V}(t) \cap \overline{\mathcal{D}^\circ[\![\tau]\!]\rho} \neq \emptyset\} \\
& = & \overline{\mathcal{T}(\mathbf{V}^\circ)} \cup \mathcal{T}(\overline{\mathcal{D}^\circ[\![\tau]\!]\rho})
\end{array}$$

which is sufficient to establish the goal. □

Proof of Lemma 5.4: Since we in the definition of the value universe **V** have used smashed product and all functions are strict, it is not difficult to see that for all $v \neq \perp$, $\mathcal{T}(v) \neq \emptyset$, i.e. Soundness Requirement 1 is satisfied.

To verify Soundness Requirement 2 we consider each of the basic type constructors.

case κ_i**.** Easy since all basic domains are assumed disjoint.

case $\tau_1 \times \tau_2$**.** Easy.

case $\tau_1 \rightarrow \tau_2$**.** To prove Property (a), assume $\mathcal{T}(\mathcal{D}^\circ[\![\tau_1]\!]\rho) \subseteq \mathcal{I}[\![\mathcal{R}[\![\tau_1]\!]]\!]\eta$ and $\mathcal{T}(\mathcal{D}^\circ[\![\tau_2]\!]\rho) \subseteq \mathcal{I}[\![\mathcal{R}[\![\tau_2]\!]]\!]\eta$.

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathcal{T}(\mathcal{D}^\circ[\![\tau_1 \rightarrow \tau_2]\!]\rho)} \\
= & \{t \in \mathit{seqs} \mid \exists f \in \mathcal{D}^\circ[\![\tau_1 \rightarrow \tau_2]\!]\rho \,.\, \mathit{elems}(t) \subseteq \mathcal{T}(\mathit{grf}(f)) \wedge \\
& \qquad \mathit{grf}(f) \subseteq \mathcal{V}(\mathit{elems}(t))\} \\
\subseteq & \{t \in \mathit{seqs} \mid \mathit{elems}(t) \subseteq \mathcal{T}(\{<x, y> \mid x \in \mathcal{D}^\circ[\![\tau_1]\!]\rho \Rightarrow y \in \mathcal{D}^\circ[\![\tau_2]\!]\rho\}) \wedge \\
& \qquad \mathit{elems}(t) \neq \emptyset\} \\
= & \{t \in \mathit{seqs} \mid \mathit{elems}(t) \subseteq \mathcal{T}(\{<x, y> \mid x \in \mathcal{D}^\circ[\![\tau_1]\!]\rho \wedge y \in \mathcal{D}^\circ[\![\tau_2]\!]\rho \vee \\
& \qquad x \in \overline{\mathcal{D}^\circ[\![\tau_1]\!]\rho} \wedge y \in \mathbf{V}^\circ\}) \wedge \\
& \qquad \mathit{elems}(t) \neq \emptyset\} \\
= & \{t \in \mathit{seqs} \mid \mathit{elems}(t) \subseteq \{p(t_1, t_2) \mid t_1 \in \mathcal{T}(\mathcal{D}^\circ[\![\tau_1]\!]\rho) \wedge t_2 \in \mathcal{T}(\mathcal{D}^\circ[\![\tau_2]\!]\rho) \vee \\
& \qquad t_1 \in \mathcal{T}(\overline{\mathcal{D}^\circ[\![\tau_1]\!]\rho}) \wedge t_2 \in \mathcal{T}(\mathbf{V}^\circ)\} \wedge \\
& \qquad \mathit{elems}(t) \neq \emptyset\}
\end{array}$$

By Lemma 5.5 we have $\mathcal{T}(\overline{\mathcal{D}^\circ[\![\tau_1]\!]\rho}) = \overline{\mathcal{T}(\mathcal{D}^\circ[\![\tau_1]\!]\rho)} \cap \mathcal{T}(\mathbf{V}^\circ)$. From the assumptions we finally get:

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathcal{T}(\mathcal{D}^\circ[\![\tau_1 \to \tau_2]\!]\rho)} \\
\subseteq & \{t \in \mathit{seqs} \mid \mathit{elems}(t) \subseteq \{p(t_1, t_2) \mid t_1 \in \mathcal{I}[\![\mathcal{R}[\![\tau_1]\!]]\!]\eta \wedge t_2 \in \mathcal{I}[\![\mathcal{R}[\![\tau]\!]]\!]\eta \vee \\
& \qquad\qquad\qquad\qquad t_1 \in \mathcal{I}[\![\overline{\mathcal{R}[\![\tau_1]\!]} * \zeta_\top]\!]\eta \wedge t_2 \in \mathcal{I}[\![\zeta_\top]\!]\eta\} \wedge \\
& \qquad \mathit{elems}(t) \neq \emptyset\} \\
= & \mathcal{I}[\![\mathcal{R}[\![\tau_1 \to \tau]\!]]\!]\eta
\end{array}$$

For Property (b) we have

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathcal{V}(\mathcal{I}[\![\mathcal{R}[\![\tau_1 \to \tau_2]\!]]\!]\eta)} \\
= & \{v \in \mathbf{V}^\circ \mid \exists t \in \mathcal{T}(v) \,.\, t \in \mathcal{I}[\![\mathcal{R}[\![\tau_1 \to \tau_2]\!]]\!]\eta\} \cup \{\bot\} \\
= & \{f \in (\mathbf{V} \to \mathbf{V})^\circ \mid \forall{<}v_1, v_2{>} \in \mathit{grf}(f) \,. \\
& \qquad \exists t_p \in \mathcal{T}({<}v_1, v_2{>}) \,. \\
& \qquad\quad t_p \in \mathcal{I}[\![p(\mathcal{R}[\![\tau_1]\!], \mathcal{R}[\![\tau_2]\!])]\!]\eta \cup \\
& \qquad\qquad \mathcal{I}[\![p(\overline{\mathcal{R}[\![\tau_1]\!]} * \zeta_\top, \zeta_\top)]\!]\eta\} \\
= & \{f \in (\mathbf{V} \to \mathbf{V})^\circ \mid \mathit{grf}(f) \subseteq \mathcal{V}(\mathcal{I}[\![p(\mathcal{R}[\![\tau_1]\!], \mathcal{R}[\![\tau_2]\!])]\!]\eta) \cup \\
& \qquad \mathcal{V}(\mathcal{I}[\![p(\overline{\mathcal{R}[\![\tau_1]\!]} * \zeta_\top, \zeta_\top)]\!]\eta)\} \\
= & \{f \in (\mathbf{V} \to \mathbf{V})^\circ \mid \mathit{grf}(f) \subseteq \mathcal{V}(\mathcal{I}[\![\mathcal{R}[\![\tau_1]\!]]\!]\eta) \times \mathcal{V}(\mathcal{I}[\![\mathcal{R}[\![\tau_2]\!]]\!]\eta) \cup \\
& \qquad \mathcal{V}(\mathcal{I}[\![\overline{\mathcal{R}[\![\tau_1]\!]} * \zeta_\top]\!]\eta) \times \mathcal{V}(\mathcal{I}[\![\zeta_\top]\!]\eta)\} \\
\subseteq & \{f \in (\mathbf{V} \to \mathbf{V})^\circ \mid \mathit{grf}(f) \subseteq \mathcal{D}^\circ[\![\tau_1]\!]\rho \times \mathcal{D}^\circ[\![\tau_2]\!]\rho \cup \\
& \qquad \overline{\mathcal{D}^\circ[\![\tau_1]\!]\rho} \times \mathbf{V}^\circ\} \\
= & \{f \in (\mathbf{V} \to \mathbf{V})^\circ \mid \mathit{grf}_{\mathit{std}}(f) \subseteq \mathcal{D}^\circ[\![\tau_1]\!]\rho \times \mathcal{D}^\circ[\![\tau_2]\!]\rho \cup \\
& \qquad \overline{\mathcal{D}^\circ[\![\tau_1]\!]\rho} \times \mathbf{V}^\circ\} \\
= & \mathcal{D}^\circ[\![\tau_1 \to \tau_2]\!]\rho
\end{array}$$

□

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399