Subtyping with Union Types, Intersection Types and Recursive Types

Flemming M. Damm

Department of Computer Science,
The Technical University of Denmark,
2800 Lyngby, Denmark
Email: fmd@id.dth.dk

Abstract. Union, intersection and recursive types are type constructions which make it possible to formulate very precise types. However, because of type checking difficulties related to the simultaneous use of these constructions, they have only found little use in programming languages. One of the problems is subtyping. We show how the subtype decision problem may be solved by an encoding into regular tree expressions, and prove soundness and completeness with respect to the classical ideal model for types [MPS86].
As a part of the work we show how the structure of values may be described by trees, and how the metric space of ideals may be related to a metric space of sets of trees. The proof techniques applied here may be of independent interest.

1 Introduction

Union and intersections types are type constructions which make it possible to formulate very precise types. This strengthens types both as a modeling tool and as a basis for automatic consistency checking (type checking).

Union types allow a partitioning of larger types which is convenient in many contexts. As an example, the integers may be partitioned into two types, the naturals and the negative integers. Also, certain meaningful expressions which otherwise are untypable become typable. As a typical example, consider the function $\texttt{lambda}\, x\,.\, \texttt{if}\, x\, \texttt{then}\, 1\, \texttt{else}\, \texttt{true}$ which has the type $\mathbf{bool} \to (\mathbf{int} \sqcup \mathbf{bool})$. The possibility of partitioning also makes it possible to define a very fine grained system of types which may form the basis for very precise type analyses. As an example, the type **bool** may be partitioned into two singleton types **true** and **false**.

Intersection types are useful in connection with functions since they allow formulation of types which characterize functions very precisely. As an example consider the if-then-else functional *cond* which may be typed:

$$cond : (\mathbf{true} \to \tau \to \mathbf{top} \to \tau) \sqcap (\mathbf{false} \to \mathbf{top} \to \tau \to \tau)$$

The more precise typing of functions, which is possible with intersection types, allows us to define more powerful type systems. Thereby more terms can be recognized as type consistent, and also the type system may form the basis for a detailed static analysis. See [FP91] and [Pie91] for interesting examples of type systems with intersection types.

There are, however, some type checking difficulties associated with the use of union and intersection types. One of the problems is the subtype decision problem, i.e., to decide whether the set of values denoted by two types are in the inclusion order.

Subtyping is closely related to inclusion polymorphism [CW85]. The idea is that an object can be used in any supertype context. As a simple example, we may want to allow application to naturals of a function taking integers as arguments. This should be allowed since the naturals can be considered a subtype of the integers. As a consequence of this view, we get that inclusion of function types is contravariant in the argument type, i.e.,

$$\frac{\sigma' \leq \sigma \quad \tau \leq \tau'}{\sigma \to \tau \leq \sigma' \to \tau'}$$

The introduction of union and intersection in a type language puts great demands on a subtyping algorithm. These type constructors are not structure forming as, e,g., products and exponentiation. This means that there is a number of type equivalence rules in addition to the ones in type languages with pure structural subtyping. Firstly, both union and intersection are associative and commutative. Secondly, there is a number of distributive laws. Union and intersection distribute over each other. Product distributes over union and intersection:

$$\tau \times (\sigma_1 \sqcup \sigma_2) = (\tau \times \sigma_1) \sqcup (\tau \times \sigma_2)$$
$$\tau \times (\sigma_1 \sqcap \sigma_2) = (\tau \times \sigma_1) \sqcap (\tau \times \sigma_2)$$

For function types, we have the following rules:

$$\sigma \to (\tau_1 \sqcap \tau_2) = (\sigma \to \tau_1) \sqcap (\sigma \to \tau_2)$$
$$(\sigma_1 \sqcup \sigma_2) \to \tau = (\sigma_1 \to \tau) \sqcap (\sigma_2 \to \tau)$$

The latter equivalence is not real distribution. But it is related to distribution because of the contravariance in the argument type of function types. Similar rules for a union in the result type and an intersection in the argument type do not hold in the general case.

Finite types result in some very special type equivalences. Consider for example the following equivalence where **unit** is a singleton type:

$$\mathbf{unit} \to (\sigma \sqcup \tau) = (\mathbf{unit} \to \sigma) \sqcup (\mathbf{unit} \to \tau)$$

This equivalence is valid because the single value in **unit** must be mapped into either a value of type σ or type τ.

A proof system for subtyping which is both sound and complete is not easily defined. To illustrate some of the difficulties, consider the following inclusion. (Product is assumed non-associative).

$$\mu t.((\tau \times t) \sqcup \mathbf{unit}) \leq \mu t.((\tau \times (\tau \times t)) \sqcup \mathbf{unit}) \sqcup \mu t.((\tau \times (\tau \times t)) \sqcup (\tau \times \mathbf{unit}))$$

Unfolding of recursive types and summand wise comparison in connection with union types are not sufficient to prove this inclusion.

Types may be considered sets of structurally similar values. We use trees to represent the structure of values. With the representation presented in this paper, it

turns out that the sets of trees which represent types, are regular. Thus types may be encoded as regular tree expressions. This yields a natural treatment of union, intersection and recursive types, and most important, we obtain a decision procedure for type inclusion. The main problem in the encoding is an appropriate treatment of functions types and the contravariance associated with it.

The finite types cause a special problem which we have not found any general solution to. In the formal work, we will therefore disregard this problem by assuming all basic types infinite. We return to the problem in Section 6. As another limitation of the approach presented here, it should be noted that we are not able deal with non-strict functions and other kinds of "lazy" values.

1.1 Related Work

Amadio and Cardelli have made considerable work on subtyping of recursive types [AC91]. They encode recursive types as regular trees and decide subtyping by an ordering on infinite trees. Their approach is, however, not immediately extensible to a type language with union and intersection types.

In a series of papers [AM91b, AM91a, AW92, AW93] Aiken, Murphy and Wimmers have developed some very interesting type systems which include both union, intersection, and recursive types. However, in connection with contravariant function types, some rough approximations are introduced. In this way a sound but incomplete system is obtained.

The work by Mishra and Reddy [MR85], Freeman and Pfenning [FP91], and Cartwright and Fagan [CF91] should also be mentioned. They all define type systems including union, intersection and recursive types, but none of them solve the subtyping problem in its full generality.

1.2 Overview

In Section 2 we present syntax and semantics of a type language including union, intersection and recursive types. We also summarize the theory underlying the ideal model of types. Section 3 presents the essential parts of the theory of regular tree expressions. Section 4 is the cornerstone of the work. Here we show how types may be encoded as regular tree expressions. In this way the subtyping problem reduces to a sublanguage problem of languages defined by regular tree expressions. Since this is known to be decidable, we obtain an algorithm for subtyping. In Section 5 we follow up on Section 4 and prove soundness and completeness of our subtyping approach. Some details are left out. These may be found in [Dam94] together with a more detailed exposition of the background theory.

2 Syntax and Semantics of Types

In this section, we will define the syntax and semantics of a small type language with union, intersection and recursive types. The semantics is a standard one viewing types as ideals [MPS86].

Let t range over type variables $\mathbf{Var}$, then the language $\mathbf{TExp}_0$ of type expressions is generated by the following grammar:

$$
\begin{aligned}
\tau ::= {} & \bot \mid \\
& \iota_1 \ldots \iota_n \mid \\
& t \mid \\
& \tau \times \tau \mid \\
& \tau \to \tau \mid \\
& \tau \sqcup \tau \mid \\
& \tau \sqcap \tau \mid \\
& \mu t.\tau
\end{aligned}
$$

The type $\bot$ denotes the empty type. The basic types are $\iota_1 \ldots \iota_n$. We shall in the following assume that all the basic types are disjoint and infinite. The former requirement is just for convenience since overlapping types may defined using the union type constructor. The latter restriction is, however, rather crucial meaning, e.g., that we cannot have the Boolean type among our basic types. We shall shortly return to the problem in Section 6. The product type $\sigma \times \tau$ is non-associative. The function type $\sigma \to \tau$ is the usual contravariant function type. All elements of a function type are assumed strict. The union and intersection types are set-theoretic, and $\mu t.\tau$ denotes the unique type σ fulfilling $\sigma = \tau[\sigma/t]$. In order to ensure unique fixed points of recursively defined types, type expressions must be *well-formed*: for all recursive types $\mu t.\tau$, all occurrences of the variable t in τ are embedded in a product or function type. The set of well-formed type expressions is denoted by **TExp**.

Before we give the formal semantics of types, we summarize some essential parts of the theory underlying the ideal model of types.

A *metric space* (D, d) [Sut75] 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* (or is a *contraction*) 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 contractions have unique fixed points:

Theorem 1 Banach. *If (D, d) is a non-empty complete metric space and $f : (D, d) \to (D, d)$ is a contraction, 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.*

A mapping $f : (D,d) \to (E,e)$ 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 $f : (D,d) \to (E,e)$ is a continuous mapping and $<s_i>_{i\geq 0}$ is a convergent sequence in (D,d) with limit $\hat{s}$, then $<f(s_i)>_{i\geq 0}$ is a convergent sequence in (E,e) with limit $f(\hat{s})$.

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

$$\mathbf{V} = \mathbf{B}_1 + \ldots + \mathbf{B}_n + (\mathbf{V} \times \mathbf{V}) + (\mathbf{V} \to \mathbf{V}) \tag{1}$$

Here $\mathbf{B}_i$ denote flat domains of basic values. These domains are assumed disjoint and infinite. By $\mathbf{V} \times \mathbf{V}$, we denote the smashed product of $\mathbf{V}$ with itself, and $\mathbf{V} \to \mathbf{V}$ is the strict continuous functions from $\mathbf{V}$ to $\mathbf{V}$. The domain operator $+$ is coalesced sum. The use of coalesced sum means that the bottom $\bot_{\mathbf{V}\to\mathbf{V}} = \lambda x . \bot$ of the function domain cannot be distinguished from $\bot_{\mathbf{V}}$.

Solutions of domain equations like (1) 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 $\mathbf{V}_n$, starting from $\mathbf{V}_0 = \{\bot\}$:

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

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 $D \to E$ the finite elements are finite lubs of *step functions*. For $d \in D^\circ$ and $d \in E^\circ$, a step function $d \Rightarrow e$ is defined by:

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

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 $\mathcal{J}(D)$.

For a finite value $v \in \mathbf{V}^\circ$ we define the *rank* of v to be the least i such that $v \in \mathbf{V}_i$. For any two sets of finite values s_1 and s_2, a *witness* is an element in their symmetric difference $s_1 \ominus s_2 = (s_1 \setminus s_2) \cup (s_2 \setminus s_1)$. The closeness $c_r(s_1,s_2)$ is the rank of the least witness. By convention, $\min\{\} = \infty$. On the basis of the closeness function, a distance function d_r between sets of finite values may be defined: $d_r(s_1,s_2) = 2^{-c(s_1,s_2)}$. By convention, $2^{-\infty} = 0$.

Theorem 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 $\hat{I} = \{v \in \mathbf{V} \mid v \text{ is in infinitely many } I_i\}$.*

Also, the subspace of order ideals $(\mathcal{J}_0(\mathbf{V}^\circ), d_r)$ is complete. If $I_1, I_2 \in \mathcal{J}(\mathbf{V})$ are distinct ideals then there exists a finite value in their symmetric difference. Hence the distance function d_r can be extended to ideals. This yields a metric space of

ideals which is also complete. We have the following close correspondence between order ideals of finite values and ideals:

Theorem 3. *The mapping $I \mapsto I^\circ$ is an order isomorphism of $(\mathcal{J}(\mathbf{V}), \subseteq)$ and $(\mathcal{J}_0(\mathbf{V}^\circ), \subseteq)$.*

This correspondence is important since it allows us to restrict our attention to the finite values of types without losing generality.

Now, the semantics of types is defined by the semantic function $\mathcal{D} : \mathbf{TExp} \to \mathbf{TEnv} \to \mathcal{J}(\mathbf{V})$ mapping well-formed type expressions into ideals. Here, $\mathbf{TEnv} = \mathbf{Var} \to \mathcal{J}(\mathbf{V})$ is the set of type environments (ranged over by ρ).

$$
\begin{aligned}
\mathcal{D}[\![\bot]\!]\rho &= \{\bot\} \\
\mathcal{D}[\![\iota_i]\!]\rho &= \mathbf{B}_i \\
\mathcal{D}[\![t]\!]\rho &= \rho(t) \\
\mathcal{D}[\![\sigma \times \tau]\!]\rho &= \{<v_1, v_2> \in \mathbf{V} \times \mathbf{V} \mid v_1 \in \mathcal{D}[\![\sigma]\!]\rho \wedge v_2 \in \mathcal{D}[\![\tau]\!]\rho\} \\
\mathcal{D}[\![\sigma \to \tau]\!]\rho &= \{f \in \mathbf{V} \to \mathbf{V} \mid \forall a \in \mathcal{D}[\![\sigma]\!]\rho \,.\, f(a) \in \mathcal{D}[\![\tau]\!]\rho\} \\
\mathcal{D}[\![\sigma \sqcup \tau]\!]\rho &= \mathcal{D}[\![\sigma]\!]\rho \cup \mathcal{D}[\![\tau]\!]\rho \\
\mathcal{D}[\![\sigma \sqcap \tau]\!]\rho &= \mathcal{D}[\![\sigma]\!]\rho \cap \mathcal{D}[\![\tau]\!]\rho \\
\mathcal{D}[\![\mu t.\tau]\!]\rho &= \mu(\lambda I \,.\, \mathcal{D}[\![\tau]\!]\rho\{I/t\})
\end{aligned}
$$

Here all meta constructions have the obvious meaning. The constructor μ is the fixed point constructor. For well-formed type expressions τ, the function $\lambda I \,.\, \mathcal{D}[\![\tau]\!]\{I/t\}$ is contractive in the metric space $(\mathcal{J}(\mathbf{V}), d_r)$ for all t. According to Banach's fixed point theorem, this means that the fixed point $\mu(\lambda I \,.\, \mathcal{D}[\![\tau]\!]\rho\{I/t\})$ is unique.

The two requirements that the basic domains are not finite and that $\lambda x \,.\, \bot$ and $\bot$ are coalesced ensures all types except the bottom type $\bot$ are infinite. If $\lambda x \,.\, \bot$ and $\bot$ are not coalesced then a singleton type $\top \to \bot$ may be defined using a universal type $\top$ which may be defined in terms of recursion.

As mentioned we shall pay special attention to finite elements. By $\mathcal{D}^\circ[\![\tau]\!]\rho$ we denote the finite elements of $\mathcal{D}[\![\tau]\!]\rho$.

3 Regular Tree Expressions

Regular tree expressions are a generalization of regular expressions. Whereas regular expressions describe sets of words, regular tree expressions describe sets of trees. A word may be considered a tree with a single branch at all nodes. We shortly summarize the parts of the theory which is important for the present work. A detailed exposition of the theory may be found in [GS84].

A ranked alphabet is a finite, non-empty set Σ of symbols together with a mapping

$$ar : \Sigma \to \mathbf{N}$$

that assigns a rank, or arity, $ar(\sigma)$ to each symbol $\sigma \in \Sigma$. By Σ_m we denote the subset of Σ with arity m.

For a given alphabet Σ and a set of variables X disjoint from Σ, the set $F_\Sigma(X)$ of ΣX-trees is defined as follows:

(i) $X \subseteq F_\Sigma(X)$,
(ii) $\sigma(t_1, \ldots, t_n) \in F_\Sigma(X)$ whenever $\sigma \in \Sigma_n$ and $t_1, \ldots, t_n \in F_\Sigma(X)$.
(iii) All ΣX-trees are obtainable by application of the above two rules.

Informally, a ΣX-tree is a labelled finite tree with node symbols Σ_m, $m > 0$ and leaf symbols $\Sigma_0 \cup X$. By H_Σ we denote the Herbrand universe of the symbols in Σ, i.e. $H_\Sigma = F_\Sigma(\emptyset)$. The *height* $hg(t)$ of a tree $t \in F_\Sigma(X)$ is the length of the longest path of t. A ΣX-forest is simply a subset of $F_\Sigma(X)$.

Just as the rank function r for finite elements of $\mathbf{V}^\circ$ served as a basis for defining the complete metric space of ideals $(\mathcal{P}(\mathbf{V}^\circ), d_r)$, the height function hg may serve as a basis for defining a complete metric space of ΣX-forests. That is, for two ΣX-forests F_1 and F_2 the closeness is defined by

$$c_{hg}(F_1, F_2) = \min\{hg(t) \mid t \in F_1 \ominus F_2\}$$

and the distance is:

$$d_{hg}(F_1, F_2) = 2^{-c_{hg}(F_1, F_2)}$$

With these definition we have

Theorem 4. *The metric space $(\mathcal{P}(F_\Sigma(X)), d_{hg})$ is complete. If $<F_i>_{i \geq 0}$ is a Cauchy sequence then its limit is $\hat{F} = \{t \in F_\Sigma(X) \mid t$ is in infinitely many $F_i\}$.*

Let X be a set of variables, and let Σ be a ranked alphabet. The set of *regular tree expressions* $\mathbf{RE}_0(\Sigma, X)$ is defined by the following grammar:

$$\begin{aligned}
\zeta ::= {} & \mathsf{bot} \mid \\
& x \mid \\
& \sigma(\zeta, \ldots, \zeta) \mid \\
& \zeta + \zeta \mid \\
& \zeta * \zeta \mid \\
& \bar{\zeta} \mid \\
& \mathsf{fix}\, x.\zeta
\end{aligned}$$

For a constructor $\sigma \in \Sigma_n$ the number of arguments must be n. Similarly to type expressions, 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.

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

In the same way as for type expression, well-formedness of regular tree expressions ensures that the application of the fixed point operator μ yields a well-defined result. Forests describable by regular tree expressions are equivalent with tree recognizers (tree automata) in a strong sense. For this reason such forests are termed ΣX-recognizable and denoted by $\mathbf{Rec}(\Sigma, X)$.

It is well known that for regular tree expressions without free variables, the inclusion problem is decidable. For regular tree expressions ζ_1 and ζ_2 with free variables, it has recently been shown that 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 Representation of Types by Regular Tree Expressions

This section consists of two parts. First we shall in a separate subsection show how trees may be used to represent the structure of values. In a second subsection, we shall generalize this representation to types, i.e. sets of values, and see that sets of trees corresponding to types constitute recognizable forests.

4.1 Values versus Trees

The principal idea of the ideal model of types, is that "the definition of ideals makes precise the notion of a type as a collection of structurally similar values" [MPS86]. We will use trees to describe this common structure of the values of a type. Since the mapping $I \mapsto I^\circ : \mathcal{J}(\mathbf{V}) \to \mathcal{J}_0(\mathbf{V}^\circ)$ is an order isomorphism of $(\mathcal{J}(\mathbf{V}), \subseteq)$ and $(\mathcal{J}_0(\mathbf{V}^\circ), \subseteq)$ (Theorem 3), we can restrict our attention to finite values without losing generality. Thus, we define two mappings $\mathcal{T} : \mathcal{P}(\mathbf{V}^\circ) \to \mathcal{P}(H_\Sigma)$ and $\mathcal{V} : \mathcal{P}(H_\Sigma) \to \mathcal{P}(\mathbf{V}^\circ)$ which relate finite values and their structure described by trees.

As a basis we introduce a set of 0-ary symbols $\kappa_1, \ldots, \kappa_n$ corresponding to the basic types $\iota_1, \ldots, \iota_n$. Thus, the structure of a value v of type ι_i is described by the tree $\kappa_i()$. This tree is unique since all basic domains are assumed disjoint. In order to describe elements of product types, we introduce a binary symbol p. The structure of a tuple $<x, y>$ of type $\sigma \times \tau$ is described by the tree $p(t_x, t_y)$ where t_x and t_y are the trees describing the structure of respectively x and y.

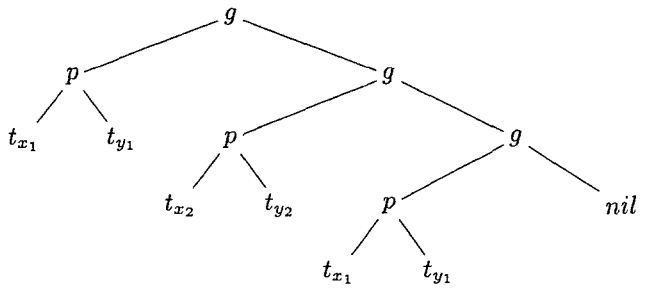
There is a problem related to representing the structure of function values by trees. Our aim is to model a function by its graph. For a function f, the graph $grf(f)$ is the set of pairs defined by:

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

Notice that there are no trivial pairs $<x, \bot>$ in a function graph. Since a set is an unordered collection, and since a tree has everything ordered, there is no unique way to represent a function graph by a tree unless some ordering is introduced. We solve this problem by representing function graphs by sets of trees. The trees corresponding to a set s are the tree encodings of sequences whose set of elements is s. Sequences are encoded as follows. Let g be a binary symbol and nil a 0-ary symbol. A sequence $<t_1, t_2, \ldots, t_n>$ is encoded as the right recursive tree $g(t_1, g(t_2, \ldots g(t_n, nil)\ldots))$. By $seqs$ we denote the set of trees representing such sequences. We will also make use

of a function *elems* which takes as argument a tree $t \in seqs$ and returns the set of elements of the sequence represented by t.

As an example, consider a function f with the graph $\{<x_1, y_1>, <x_2, y_2>\}$. If t_{x_1}, t_{y_1}, t_{x_2} and t_{y_2} are trees representing respectively x_1, y_1, x_2 and y_2, then one of the trees representing f is:



With these prerequisites, we may now formally define the mapping from values to sets of trees. The function $\mathcal{T}$ is first defined for values:

$$
\begin{aligned}
\mathcal{T}(\bot) &= \emptyset \\
\mathcal{T}(b_{ij} : \mathbf{B}_i) &= \{\kappa_i\} \\
\mathcal{T}(<v_1, v_2> : (\mathbf{V} \times \mathbf{V})^\circ) &= \{p(t_1, t_2) \mid t_1 \in \mathcal{T}(v_1) \wedge t_2 \in \mathcal{T}(v_2)\} \\
\mathcal{T}(f : (\mathbf{V} \to \mathbf{V})^\circ) &= \{t \in seqs \mid \\
&\qquad \forall t_p \in elems(t) . \\
&\qquad\quad \exists <v_1, v_2> \in grf(f) . t_p \in \mathcal{T}(<v_1, v_2>) \wedge \\
&\qquad \forall <v_1, v_2> \in grf(f) . \\
&\qquad\quad \exists t_p \in elems(t) . t_p \in \mathcal{T}(<v_1, v_2>)\}
\end{aligned}
$$

For any sequence representing a function f, each element of the sequence must represent a pair in the function graph (the first universal quantification), and each element in the function graph must be represented by an element in the sequence (the second universal quantification). Also note that since we have used coalesced sum in the definition of the value universe, we have $\lambda x . \bot = \bot$ and thus $\mathcal{T}(\lambda x . \bot)$ is the empty set $\emptyset$ and not the singleton set $\{nil()\}$. Therefore, all sequence representations of functions are non-empty.

On the basis of the definition of $\mathcal{T}$, a precise definition of *structural equivalence* may be given: two values are structural equivalent whenever they are represented by the same set of trees.

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

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

It is an immediate consequence of this definition that $\mathcal{T}$ is monotonic with respect to inclusion.

The set of all finite values $\mathbf{V}^\circ$ is mapped to the set of trees denoted by the regular tree expression:

$$\zeta_\mathcal{T} = \mathrm{fix}\, t.\kappa_1 + \ldots + \kappa_n + p(t,t) + \mathrm{fix}\, s.(g(p(t,t), s) + g(p(t,t), nil))$$

Note the encoding of non-empty sequences.

Lemma 5. *For all forest environments η,*

$$\mathcal{T}(\mathbf{V}^\circ) = \mathcal{I}[\![\zeta_\top]\!]\eta$$

The problem previously mentioned regarding finite basic domains is related to the definition of $\zeta_\top$. Having basic domains which are finite, the sets of trees denoted by $\zeta_\top$ would be a proper superset of $\mathcal{T}(\mathbf{V}^\circ)$. As an example, consider the tree $t = g(p(\kappa_1, \kappa_2), g(p(\kappa_1, \kappa_3), nil))$. If $\mathbf{B}_1$ only holds a single value (except for $\bot$) then there will be no value $v \in \mathbf{V}^\circ$ such that $t \in \mathcal{T}(v)$. Similar problems occur when $\lambda x \,.\, \bot$ and $\bot$ are not coalesced since then a singleton type $\top \to \bot$ (represented by $nil()$) may be defined. If some of the basic domains are finite or $\lambda x \,.\, \bot$ and $\bot$ are not coalesced, then it is not possible to formulate a regular tree expression which denotes exactly the set of trees $\mathcal{T}(\mathbf{V}^\circ)$.

Next we define a mapping $\mathcal{V}$ going in the other direction, i.e. mapping which for a tree t yields the values which have the structure described by t.

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

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.

$$\mathcal{V}(F) = \bigcup\{\mathcal{V}(t) \mid t \in F\} \cup \{\bot\}$$

Like $\mathcal{T}$, $\mathcal{V}$ is monotonic with respect to inclusion. From the definition of $\mathcal{T}$ it is easy to infer the following equalities:

$$\begin{aligned} \mathcal{V}(F) &= \{v \in \mathbf{V}^\circ \mid \exists t \in F \,.\, t \in \mathcal{T}(v)\} \cup \{\bot\} \\ &= \{v \in \mathbf{V}^\circ \mid \exists t \in \mathcal{T}(v) \,.\, t \in F\} \cup \{\bot\} \end{aligned} \tag{2}$$

As a basis for formalization of properties related to the correspondence between types, values, and trees, we define the notion of agreeability of forest environments and type environments.

Definition 6 Agreement. A type environments ρ and a forest environment η agree if for all variables $t \in \mathbf{Var}$:

$$\begin{aligned} \rho(t)^\circ &= \mathcal{V}(\eta(t)) \quad \text{and} \\ \eta(t) &= \mathcal{T}(\rho(t)^\circ) \end{aligned}$$

A type environment ρ is *agreeable* if there exists a forest environment η agreeing with ρ. Informally this means that no sets of values in the range of an agreeable type environment fulfills an invariant which cannot be expressed by trees as representations of sets of values. In other words, all sets of values in the range of an agreeable type environment are closed under structural equivalence.

As an example of a set of values which does not meet this requirement, consider the set of the pairs

$$P = \{<x, y> \in \mathbf{B}_i \times \mathbf{B}_i \mid x = y\} \cup \{\bot\}$$

The tree representation of this set is $p(\kappa_i, \kappa_i)$. It easy to see that $\mathcal{V}(\mathcal{T}(P)) = \mathbf{B}_i \times \mathbf{B}_i$ meaning that the representation $p(\kappa_i, \kappa_i)$ "forgets" the invariant that $x = y$ for all $<x, y> \in P$.

The closure under structural equivalence of sets of values in the range of agreeable type environment, is generalizable to types:

Lemma 7. *For all types τ and all agreeable type environments ρ ranging over the free variables of τ:*

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

The lemma follows immediately from the following two lemmas:

Lemma 8. *Let τ be any type, and ρ any agreeable type environment. Let $v \neq \perp$ be any finite value, i.e. $v \in \mathbf{V}^\circ$. Let $v' \neq \perp$ be any value structurally equivalent with v, i.e. $v' \in \mathcal{V}(\mathcal{T}(v))$. Then $v \in \mathcal{D}^\circ[\![\tau]\!]\rho$ iff $v' \in \mathcal{D}^\circ[\![\tau]\!]\rho$.*

Proof. See appendix. □

Lemma 9. *Let $v \in \mathbf{V}^\circ$ be any finite value. Then for all $t \in \mathcal{T}(v)$, $v \in \mathcal{V}(t)$.*

Proof. Follows immediately from the definition of $\mathcal{V}$. □

4.2 Types versus Recognizable Forests

As a last step in our development, we shall show how the sets of trees corresponding to types constitute recognizable forests, i.e. forests describable by regular tree expressions. The monotonicity with respect to inclusion of the mapping $\mathcal{T}$ from sets of values to recognizable forests gives completeness of subtyping based on inclusion of recognizable forests. Soundness follows from the monotonicity of $\mathcal{V}$ and the close correspondence between trees and values (Lemma 7). Thus, we are aiming at a function $\mathcal{R} : \mathbf{TExp} \to \mathbf{RE}(\Sigma, \mathbf{Var})$ transforming type expressions into regular tree expressions such that the following two diagrams commute:

$$\begin{array}{ccccc}
\mathbf{TExp} & \xrightarrow{\mathcal{D}} & \mathcal{J}(\mathbf{V}) & \xrightarrow{I \mapsto I^\circ} & \mathcal{J}_0(\mathbf{V}^\circ) \\
\downarrow{\scriptstyle \mathcal{R}} & & & & \downarrow{\scriptstyle \mathcal{T}} \\
\mathbf{RE}(\Sigma, \mathbf{Var}) & & \xrightarrow[\mathcal{I}]{} & & \mathbf{Rec}(\Sigma, \emptyset)
\end{array} \tag{3}$$

$$\begin{array}{ccccc}
\mathbf{TExp} & \xrightarrow{\mathcal{D}} & \mathcal{J}(\mathbf{V}) & \xrightarrow{I \mapsto I^\circ} & \mathcal{J}_0(\mathbf{V}^\circ) \\
\downarrow{\scriptstyle \mathcal{R}} & & & & \uparrow{\scriptstyle \mathcal{V}} \\
\mathbf{RE}(\Sigma, \mathbf{Var}) & & \xrightarrow[\mathcal{I}]{} & & \mathbf{Rec}(\Sigma, \emptyset)
\end{array} \tag{4}$$

The function $\mathcal{R} : \mathbf{TExp} \to \mathbf{RE}(\Sigma, \mathbf{Var})$ may be defined as follows:

$$
\begin{aligned}
\mathcal{R}[\![\bot]\!] &= \mathsf{bot} \\
\mathcal{R}[\![\iota_i]\!] &= \kappa_i \\
\mathcal{R}[\![t]\!] &= t \\
\mathcal{R}[\![\sigma \times \tau]\!] &= p(\mathcal{R}[\![\sigma]\!], \mathcal{R}[\![\tau]\!]) \\
\mathcal{R}[\![\sigma \to \tau]\!] &= \mathsf{fix}\, t.g(\zeta_p, t) + g(\zeta_p, nil) \\
&\qquad \text{where } \zeta_p = p(\mathcal{R}[\![\sigma]\!], \mathcal{R}[\![\tau]\!]) + p(\overline{\mathcal{R}[\![\sigma]\!]} * \zeta_\top, \zeta_\top) \text{ and} \\
&\qquad t \notin FV(\zeta_p). \\
\mathcal{R}[\![\sigma \sqcup \tau]\!] &= \mathcal{R}[\![\sigma]\!] + \mathcal{R}[\![\tau]\!] \\
\mathcal{R}[\![\sigma \sqcap \tau]\!] &= \mathcal{R}[\![\sigma]\!] * \mathcal{R}[\![\tau]\!] \\
\mathcal{R}[\![\mu t.\tau]\!] &= \mathsf{fix}\, t.\mathcal{R}[\![\tau]\!]
\end{aligned}
$$

In the most cases the transformation is straightforward and follows in a natural way from the definition of $\mathcal{T}$. A basic type ι_i is transformed into the regular tree expression κ_i. A product type $\sigma \times \tau$ is transformed into an expression denoting the trees of the form $p(t_\sigma, t_\tau)$ where t_σ and t_τ are trees corresponding to the factor types σ and τ. Union and intersection of types are transformed into union and intersection on regular tree expressions. This is natural since in both worlds union and intersection are set theoretic.

Turning to functions, things becomes more complex. The structural similarity of elements of a function type is not as simple as for basic types and product types. Consider for example the type $\mathbf{int} \to \mathbf{bool}$. For a function f of type $\mathbf{int} \to \mathbf{bool}$ we have that for all $<x, y>$ in the graph of f, if x is of type **int** then y is of type **bool**. But, if x is not of type **int** then there is no restriction on the type of y. Thus both $(1 \Rightarrow \mathsf{true}) \sqcup (\mathsf{true} \Rightarrow 1)$ and $(1 \Rightarrow \mathsf{true}) \sqcup (\mathsf{true} \Rightarrow \mathsf{true})$ are of type $\mathbf{int} \to \mathbf{bool}$. (Here $\sqcup$ is the lattice theoretic lub operator). This shows that even though we consider types as sets of structurally similar values, the values need not be structurally equivalent. For function types, the structural similarity is maybe best understood by considering the negative information about the structure of elements: for all functions f of type $\sigma \to \tau$, there is no pair $<x, y>$ in the graph of f such that x has type σ and y does *not* have type τ. The transformation of function types has to take the negative information into account. This is the intuition behind the function graph elements described by the trees $p(\overline{\mathcal{R}[\![\sigma]\!]} * \zeta_\top, \zeta_\top)$. This says that a function f of type $\sigma \to \tau$ may map a value which is not in the domain type into any value. Also note that sequences describing function graphs must be non-empty since $\lambda x.\bot$ is coalesced with $\bot$.

Recursive types are transformed into corresponding recursive regular tree expressions. This transformation relates to the fact that the mapping $\mathcal{T}$ from trees to values are continuous as a mapping between metric spaces and thus preserve limits.

5 Soundness and Completeness

In this section which is rather technical, we formalize and prove the properties which lead us to the main result of this paper: soundness and completeness of subtyping

based on the encoding of types described in the previous section.

The central idea of our approach is the relation of finite values and trees. For this reason properties of the mappings $\mathcal{T}$ and $\mathcal{V}$ are central to the proof of soundness and completeness. The first lemma which states continuity of $\mathcal{T}$, allows us to reason about recursive types since continuous functions preserve limits.

Lemma 10 Continuity of $\mathcal{T}$. *The function $\mathcal{T}$ from the metric space $(\mathcal{P}(\mathbf{V}^\circ), d_r)$ to the metric space (H_Σ, d_{hg}) is continuous.*

Proof. In terms of closeness the function $\mathcal{T} : \mathcal{P}(\mathbf{V}^\circ) \to H_\Sigma$ is continuous in $I_0 \in \mathcal{P}(\mathbf{V}^\circ)$ if for all $m > 0$ there exists an $n > 0$ such that for all $I \in \mathcal{P}(\mathbf{V}^\circ)$, $c(\mathcal{T}(I_0), \mathcal{T}(I)) \leq m \Rightarrow c(I, I_0) \leq n$. We prove that for all $I_0, I \in \mathcal{P}(\mathbf{V}^\circ)$ and all $m > 0$, $c(\mathcal{T}(I_0), \mathcal{T}(I)) \leq m \Rightarrow c(I, I_0) \leq m$. The important observation is that

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

for all values $v \in \mathbf{V}^\circ$. This may proved by the by induction on (the rank of) v. Now, let $I_0, I \in \mathcal{P}(\mathbf{V}^\circ)$ be any sets of values, and $m > 0$. Assume $c(\mathcal{T}(I_0), \mathcal{T}(I)) \leq m$. That is, there exists a witness t_w of $\mathcal{T}(I_0)$ and $\mathcal{T}(I)$ such that $hg(t_w) \leq m$. It follows from the element wise definition of $\mathcal{T}$ that the elements of the set $\{v \mid t_w \in \mathcal{T}(v)\} \cap (I_0 \ominus I)$ are witnesses of I_0 and I. From the above it follows $\max\{r(v) \mid t_w \in \mathcal{T}(v)\} \leq m$, i.e. $c(I_0, I) \leq m$. □

Lemma 11. *Let $v_1, v_2 \in \mathbf{V}^\circ$ be any values. If $\mathcal{T}(v_1) \cap \mathcal{T}(v_2) \neq \emptyset$ then for all types τ and agreeable type environments ρ, $v_1 \in \mathcal{D}^\circ[\![\tau]\!]\rho$ iff $v_2 \in \mathcal{D}^\circ[\![\tau]\!]\rho$*

Proof. Assume $v_1 \in \mathcal{D}^\circ[\![\tau]\!]\rho$ but $v_2 \notin \mathcal{D}^\circ[\![\tau]\!]\rho$. By Lemma 8 we have $\mathcal{V}(\mathcal{T}(v_1)) \subseteq \mathcal{D}^\circ[\![\tau]\!]\rho$ and $\mathcal{V}(\mathcal{T}(v_2)) \cap \mathcal{D}^\circ[\![\tau]\!]\rho = \{\bot\}$. Thus $\mathcal{V}(\mathcal{T}(v_1)) \cap \mathcal{V}(\mathcal{T}(v_2)) = \{\bot\}$. Since $\mathcal{V}$ is monotonic with respect to inclusion, $\mathcal{V}(\mathcal{T}(v_1) \cap \mathcal{T}(v_2)) \subseteq \{\bot\}$. From Lemma 9 it follows $\mathcal{T}(v_1) \cap \mathcal{T}(v_2) = \emptyset$. □

Informally, Lemma 11 says that if two values generate equal trees, i.e. they have the same structure, then the values are of the same type. In other words, if two values have different types then they are structurally different. In view of the characterization of a type as a notion for a collection of structurally similar values, the lemma says that trees are a sufficient precise representation of the structure of values. On the basis of Lemma 11 we can derive the following two corollaries:

Corollary 12. *For all types σ, τ and agreeable type environments ρ,*

$$\mathcal{T}(\mathcal{D}^\circ[\![\sigma \sqcap \tau]\!]\rho) = \mathcal{T}(\mathcal{D}^\circ[\![\sigma]\!]\rho) \cap \mathcal{T}(\mathcal{D}^\circ[\![\tau]\!]\rho)$$

Proof. From the monotonicity of $\mathcal{T}$ it follows $\mathcal{T}(\mathcal{D}^\circ[\![\sigma \sqcap \tau]\!]\rho) \subseteq \mathcal{T}(\mathcal{D}^\circ[\![\sigma]\!]\rho) \cap \mathcal{T}(\mathcal{D}^\circ[\![\tau]\!]\rho)$. Assume $\mathcal{T}(\mathcal{D}^\circ[\![\sigma \sqcap \tau]\!]\rho) \subset \mathcal{T}(\mathcal{D}^\circ[\![\sigma]\!]\rho) \cap \mathcal{T}(\mathcal{D}^\circ[\![\tau]\!]\rho)$. Then there exists a tree $t \in \mathcal{T}(\mathcal{D}^\circ[\![\sigma]\!]\rho) \cap \mathcal{T}(\mathcal{D}^\circ[\![\tau]\!]\rho)$ such that $t \notin \mathcal{T}(\mathcal{D}^\circ[\![\sigma \sqcap \tau]\!]\rho)$. Also, there exist two values $v_1 \in \mathcal{D}^\circ[\![\sigma]\!]\rho \setminus \mathcal{D}^\circ[\![\tau]\!]\rho$ and $v_2 \in \mathcal{D}^\circ[\![\tau]\!]\rho \setminus \mathcal{D}^\circ[\![\sigma]\!]\rho$ such that $t \in \mathcal{T}(v_1) \cap \mathcal{T}(v_2)$. From Lemma 8 it follows that $\mathcal{V}(\mathcal{T}(v_1)) \subseteq \mathcal{D}^\circ[\![\sigma]\!]\rho \setminus \mathcal{D}^\circ[\![\tau]\!]\rho$ and $\mathcal{V}(\mathcal{T}(v_2)) \subseteq \mathcal{D}^\circ[\![\tau]\!]\rho \setminus \mathcal{D}^\circ[\![\sigma]\!]\rho$. Since $\mathcal{V}$ is monotonous, we have $\mathcal{T}(v_1) \cap \mathcal{T}(v_2) = \emptyset$. This leads, however, to a contradiction since we have already proved $t \in \mathcal{T}(v_1) \cap \mathcal{T}(v_2)$. □

Corollary 13. *For all types τ and agreeable type environments ρ,*

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

where $\overline{\mathcal{D}^\circ[\![\tau]\!]\rho} = \mathbf{V}^\circ \setminus \mathcal{D}^\circ[\![\tau]\!]\rho$.

Proof. Follows from Lemma 8 and monotonicity of $\mathcal{T}$. □

We are now able to prove the following lemma which is the cornerstone in the proofs of the soundness and completeness theorems.

Lemma 14. *For any type expression τ, any type environment ρ, and any forest environment η agreeing with ρ, we have*

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

That is, the diagram (3) commutes.

Proof. The proof is by induction on the structure of τ. We show the most difficult cases:

case $\tau \equiv \sigma \to \upsilon$.

$$\begin{aligned}
&\mathcal{T}(\mathcal{D}^\circ[\![\sigma \to \upsilon]\!]\rho) \\
&\quad = \{t \in seqs \mid \\
&\qquad \exists f \in \mathcal{D}^\circ[\![\sigma \to \upsilon]\!]\rho \setminus \{\bot\} . \\
&\qquad\quad \forall t_p \in elems(t) . \exists <v_1, v_2> \in grf(f) . t_p \in \mathcal{T}(<v_1, v_2>) \wedge \\
&\qquad\quad \forall <v_1, v_2> \in grf(f) . \exists t_p \in elems(t) . t_p \in \mathcal{T}(<v_1, v_2>)\}
\end{aligned}$$

The universal quantification $\forall t_p \in elems(t) \ldots$. gives an upper limit on the set of elements of t, and the existential quantification $\forall <v_1, v_2> \in grf(f) \ldots$. gives a lower limit on the set of elements of t. Since the set $\mathcal{D}^\circ[\![\sigma \to \upsilon]\!]\rho$ is downward closed, the requirement $\forall <v_1, v_2> \in grf(f) \ldots$. is always fulfilled. However, as $f \neq \bot$ the latter requirement means that the set of elements of t should be non-empty.

$$\begin{aligned}
&\mathcal{T}(\mathcal{D}^\circ[\![\sigma \to \upsilon]\!]\rho) \\
&\quad = \{t \in seqs \mid \\
&\qquad \exists f \in \mathcal{D}^\circ[\![\sigma \to \upsilon]\!]\rho \setminus \{\bot\} . \\
&\qquad\quad \forall t_p \in elems(t) . \exists <v_1, v_2> \in grf(f) . t_p \in \mathcal{T}(<v_1, v_2>) \wedge \\
&\qquad\quad elems(t) \neq \emptyset\}
\end{aligned}$$

For any function $f \in \mathcal{D}^\circ[\![\sigma \to \upsilon]\!]\rho \setminus \{\bot\}$, we have $grf(f) \subseteq \{<v_1, v_2> \mid v_1 \in \mathcal{D}^\circ[\![\sigma]\!]\rho \Rightarrow v_2 \in \mathcal{D}^\circ[\![\tau]\!]\rho\}$. On the other hand, not every subset of $\{<v_1, v_2> \mid v_1 \in \mathcal{D}^\circ[\![\sigma]\!]\rho \Rightarrow v_2 \in \mathcal{D}^\circ[\![\tau]\!]\rho\}$ is a function graph. For a function f, we have the invariant that for all $<v_1, v_2>, <v_1', v_2'> \in grf(f)$, $v_1 = v_1'$ implies $v_2 = v_2'$. However, if s is a finite subset of $\{<v_1, v_2> \mid v_1 \in \mathcal{D}^\circ[\![\sigma]\!]\rho \Rightarrow v_2 \in \mathcal{D}^\circ[\![\tau]\!]\rho\}$, then all pairs $<v_1, v_2>$ which obscure this function invariant can be replaced with a new pair $<v_1', v_2'>$ which is type equivalent (structurally equivalent) with $<v_1, v_2>$ but does not obscure the invariant. This is possible because all types are infinite.

$$
\begin{aligned}
&\mathcal{T}(\mathcal{D}^\circ[\![\sigma \to v]\!]\rho) \\
&= \{t \in seqs \mid \\
&\qquad \forall t_p \in elems(t)\,. \\
&\qquad\quad \exists <v_1, v_2> \in \{<v_1, v_2> \mid v_1 \in \mathcal{D}^\circ[\![\sigma]\!]\rho \Rightarrow v_2 \in \mathcal{D}^\circ[\![\tau]\!]\rho\}\,. \\
&\qquad\qquad t_p \in \mathcal{T}(<v_1, v_2>) \wedge \\
&\qquad elems(t) \neq \emptyset\} \\
&= \{t \in seqs \mid \\
&\qquad elems(t) \subseteq \mathcal{T}(\{<v_1, v_2> \mid v_1 \in \mathcal{D}^\circ[\![\sigma]\!]\rho \Rightarrow v_2 \in \mathcal{D}^\circ[\![\tau]\!]\rho\}) \wedge \\
&\qquad elems(t) \neq \emptyset\} \\
&= \{t \in seqs \mid \\
&\qquad elems(t) \subseteq \mathcal{T}(\{<v_1, v_2> \mid v_1 \in \mathcal{D}^\circ[\![\sigma]\!]\rho \wedge v_2 \in \mathcal{D}^\circ[\![\tau]\!]\rho \vee \\
&\qquad\qquad\qquad\qquad v_1 \in \overline{\mathcal{D}^\circ[\![\sigma]\!]\rho} \wedge v_2 \in \mathbf{V}^\circ\}) \wedge \\
&\qquad elems(t) \neq \emptyset\} \\
&= \{t \in seqs \mid \\
&\qquad elems(t) \subseteq \{p(t_1, t_2) \mid t_1 \in \mathcal{T}(\mathcal{D}^\circ[\![\sigma]\!]\rho) \wedge t_2 \in \mathcal{T}(\mathcal{D}^\circ[\![\tau]\!]\rho) \vee \\
&\qquad\qquad\qquad\qquad t_1 \in \mathcal{T}(\overline{\mathcal{D}^\circ[\![\sigma]\!]\rho}) \wedge t_2 \in \mathcal{T}(\mathbf{V}^\circ)\} \wedge \\
&\qquad elems(t) \neq \emptyset\}
\end{aligned}
$$

By Corollary 13 we have $\mathcal{T}(\overline{\mathcal{D}^\circ[\![\sigma]\!]\rho}) = \overline{\mathcal{T}(\mathcal{D}^\circ[\![\sigma]\!]\rho)} \cap \mathcal{T}(\mathbf{V}^\circ)$. From the induction hypothesis and Lemma 5, we finally get:

$$
\begin{aligned}
&\mathcal{T}(\mathcal{D}^\circ[\![\sigma \to v]\!]\rho) \\
&= \{t \in seqs \mid \\
&\qquad elems(t) \subseteq \{p(t_1, t_2) \mid t_1 \in \mathcal{I}[\![\mathcal{R}[\![\sigma]\!]]\!]\eta \wedge t_2 \in \mathcal{I}[\![\mathcal{R}[\![\tau]\!]]\!]\eta \vee \\
&\qquad\qquad\qquad\qquad t_1 \in \mathcal{I}[\![\overline{\mathcal{R}[\![\sigma]\!]} * \zeta_\top]\!]\eta \wedge t_2 \in \mathcal{I}[\![\zeta_\top]\!]\eta\} \wedge \\
&\qquad elems(t) \neq \emptyset\} \\
&= \mathcal{I}[\![\mathcal{R}[\![\sigma \to \tau]\!]]\!]\eta
\end{aligned}
$$

case $\tau \equiv \sigma \sqcap v$. This follows immediately from Corollary 12 and the induction hypothesis.

case $\tau \equiv \mu t.\sigma$. $\mathcal{T}(\mathcal{D}^\circ[\![\mu t.\sigma]\!]\rho) = \mathcal{T}(\mu(\lambda I\,.\,\mathcal{D}^\circ[\![\sigma]\!]\rho\{I/t\}))$. (Here we have made use of the fact that the mapping $I \mapsto I^\circ$ is continuous and that $\mathcal{D}^\circ[\![\sigma]\!]\rho\{I/t\} = \mathcal{D}^\circ[\![\sigma]\!]\rho\{I^\circ/t\}$). Let $D \equiv \lambda I\,.\,\mathcal{D}^\circ[\![\sigma]\!]\rho\{I/t\}$ and let $I_0 \in \mathcal{J}_0(\mathbf{V}^\circ)$ be any order ideal such that $\mathcal{V}(\mathcal{T}(I_0)) = I_0$. This requirement on I_0 ensures that $\rho\{D^n(I_0)/t\}$ and $\eta\{\mathcal{T}(D^n(I_0))/t\}$ agree. The above fixed point $\mu(D)$ is the limit of the chain $<D^n(I_0)>_{n\geq 0}$. Since $\mathcal{T}$ is continuous, $\mathcal{T}(\mu(D))$ is the limit of the chain $<\mathcal{T}(D^n(I_0))>_{n\geq 0}$. Let $T \equiv \lambda F\,.\,\mathcal{I}[\![\mathcal{R}[\![\sigma]\!]]\!]\eta\{F/t\}$. Then for all n, $\mathcal{T}(D^n(I_0)) = T^n(\mathcal{T}(I_0))$. This is proved by induction on n. The base case $n = 0$ is trivial. Next assume $\mathcal{T}(D^n(I_0)) = T^n(\mathcal{T}(I_0))$. Then

$$
\begin{aligned}
&\mathcal{T}(D^{n+1}(I_0)) \\
&= \mathcal{T}(\mathcal{D}^\circ[\![\sigma]\!]\rho\{D^n(I_0)/t\}) \\
&= \mathcal{I}[\![\mathcal{R}[\![\sigma]\!]]\!]\eta\{\mathcal{T}(D^n(I_0))/t\} \text{ by the outer induction hypothesis} \\
&= \mathcal{I}[\![\mathcal{R}[\![\sigma]\!]]\!]\eta\{T^n(\mathcal{T}(I_0))/t\} \text{ by the induction hypothesis} \\
&= T^{n+1}(\mathcal{T}(I_0))
\end{aligned}
$$

Since the limit of the chain $<T^n(I_0)>_{n\geq 0}$ is $\mu(T)$, we have proved $\mu(T) =$

$\mathcal{T}(\mu(D))$. That is

$$\begin{aligned}
&\mathcal{T}(\mathcal{D}^\circ[\![\mu t.\sigma]\!]\rho) \\
&= \mathcal{T}(\mu(D)) \\
&= \mu(T) \\
&= \mu(\lambda F\,.\,\mathcal{I}[\![\mathcal{R}[\![\sigma]\!]]\!]\eta\{F/t\}) \\
&= \mathcal{I}[\![\mathcal{R}[\![\mu t.\sigma]\!]]\!]\eta
\end{aligned}$$

□

With this lemma and Lemma 7 it is now easy to prove:

Lemma 15. *For any type expression τ, any agreeable type environment ρ, and any forest environment η agreeing with ρ, we have*

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

That is, the diagram (4) commutes.

On the basis of the latter two lemmas, the monotonicity of $\mathcal{T}$ and $\mathcal{V}$ with respect to inclusion, and the isomorphism of $(\mathcal{J}(\mathbf{V}), \subseteq)$ and $(\mathcal{J}_0(\mathbf{V}^\circ), \subseteq)$, it is easy to prove the following two theorems which state respectively soundness and completeness of our subtyping approach.

Theorem 16 Soundness of subtyping. *Let $\sigma, \tau \in$ **TExp** be any types. If for all forests environments η, $\mathcal{I}[\![\mathcal{R}[\![\sigma]\!]]\!]\eta \subseteq \mathcal{I}[\![\mathcal{R}[\![\tau]\!]]\!]\eta$ then there exists an (agreeable) type environment ρ such that $\mathcal{D}[\![\sigma]\!]\rho \subseteq \mathcal{D}[\![\tau]\!]\rho$.*

Theorem 17 Completeness of subtyping. *Let $\sigma, \tau \in$ **TExp** be any types. If $\mathcal{D}[\![\sigma]\!]\rho \subseteq \mathcal{D}[\![\tau]\!]\rho$ for all (agreeable) type environments, then there exists a forest environment η such that $\mathcal{I}[\![\mathcal{R}[\![\sigma]\!]]\!]\eta \subseteq \mathcal{I}[\![\mathcal{R}[\![\tau]\!]]\!]\eta$.*

6 Concluding Remarks

We have seen how the subtyping problem may be transformed to an inclusion problem on regular tree expressions by encoding types as regular tree expressions. This may be considered a step towards supporting the use of union, intersection and recursive types in practical programming and specification languages. The results presented here allow formulation of new powerful type systems which are decidable.

It is on the agenda for future work to investigate whether the ideas presented here may serve as a basis for solving sets of type inclusion constraints. This problem is closely related to type inference [AW93]. A type inclusion constraint has the form $\sigma \leq \tau$ where σ and τ are types which may contain free variables. A solution to a set of type inclusion constraints is a substitution of the free variables such that all constraints are satisfied. For regular tree expressions, it is already known how to solve sets of inclusions constraints [AW92]. Thus, what is needed is a relation of solutions for systems of type inclusion constraints and solutions of systems of constraints on regular tree expressions.

There are, however, also some limitations to our subtyping approach. As mentioned, there is a problem associated with finite types. This may be solved to some extent by a special treatment of functions having finite argument types but it is an open question whether the problem is solvable in its full generality. It appears likely that the introduction of finite types does not affect the soundness result. However, the present proof of soundness relies on lemmas assuming that types are infinite.

The set of type constructions introduced here is limited. Some type constructions, e.g., list types, are easily added to the type language without affecting the basic ideas in the formal work. However, the introduction of some type constructions, e.g., lazy function types, is not trivial, and if possible, it requires thorough modifications in the formal work.

Finally, there may be a complexity problem. Deciding inclusion of regular tree expressions requires time exponential to the size of the expressions. Thus subtyping may also require exponential time. However, for some type systems, the types introduced in practical programs may have a sufficiently limited size making type checking with this approach feasible.

Acknowledgement

I am very thankful to Bo Stig Hansen. His interest, comments and response have been invaluable for the work presented here.

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. Extended abstract. Full version in [AC92].

[AC92] Roberto M. Amadio and Luca Cardelli. *Subtyping Recursive Types.* Technical Report 133, INRIA, Institut National de Recherche en Informatique et en Automatique, Lorraine, February 1992.

[AM91a] 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.

[AM91b] Alexander Aiken and Brian M. Murphy. Static type inference in a dynamically typed language. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages (POPL'91)*, pages 279–290, ACM Press, 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, 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.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, 1985.

[Dam94] Flemming M. Damm. *Subtyping with Union Types, Intersection Types and Recursive Types.* Technical Report, Department of Computer Science, The Technical University of Denmark, 1994. Forthcoming.

[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, IEEE Computer Society Press, 1993.

[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.

[Pie91] Benjamin C. Pierce. *Programming with Intersection Types, Union Types and Polymorphism.* Technical Report CMU-CS-91-106, Carnegie Mellon University, 1991.

[Sut75] W.A. Sutherland. *Introduction to metric and topological spaces.* Oxford University Press, 1975.

A Proofs

Proof of Lemma 8. We consider the types not containing recursion and those containing recursion in two separate cases. The proof of the latter case relies on the proof of the first case. First, let τ be any type expression not containing recursion. The proof is by induction on τ. In each induction step we make a case on analysis on the value v. We will make implicit use of the fact that for all values $v \in \mathbf{V}^\circ$, $\mathcal{T}(v) = \emptyset$ iff $v = \bot$. Notice that it is a simple consequence of the definition of $\mathcal{T}$ and $\mathcal{V}$ that $\mathcal{V} \circ \mathcal{T}$ maps basic values to set of basic values, tuples to set of tuples, and functions to set of functions. The most difficult cases of the induction proof are:

case $\tau \equiv \sigma \times \tau$. The only interesting case is $v \in (\mathbf{V} \times \mathbf{V})^\circ$ Assume $v \in (\mathbf{V} \times \mathbf{V})^\circ$, i.e. $v \equiv <a, b>$ for some $a, b \in \mathbf{V}^\circ$.

$$\mathcal{T}(<a,b>) = \{p(t_1, t_2) \mid t_1 \in \mathcal{T}(a) \wedge t_2 \in \mathcal{T}(b)\}$$

Applying $\mathcal{V}$ we get:

$$\begin{aligned}
&\mathcal{V}(\mathcal{T}(<a,b>)) \\
&= \{<v_1, v_2> \in (\mathbf{V} \times \mathbf{V})^\circ \mid \exists t_1 \in \mathcal{T}(v_1) \,.\, t_1 \in \mathcal{T}(a) \wedge \\
&\qquad\qquad \exists t_2 \in \mathcal{T}(v_2) \,.\, t_2 \in \mathcal{T}(b)\} \\
&= \{<v_1, v_2> \in (\mathbf{V} \times \mathbf{V})^\circ \mid v_1 \in \mathcal{V}(\mathcal{T}(a)) \wedge v_2 \in \mathcal{V}(\mathcal{T}(b))\}
\end{aligned}$$

The last equality follows from equation (2). Now it follows immediately from the induction hypothesis that for all $<a', b'> \in \mathcal{V}(\mathcal{T}(<a, b>))$, $<a, b> \in \mathcal{D}^\circ[\![\sigma_1 \times \sigma_2]\!]\rho$ iff $<a', b'> \in \mathcal{D}^\circ[\![\sigma_1 \times \sigma_2]\!]\rho$.

case $\tau \equiv \sigma \to v$. The only interesting case is $v \in (\mathbf{V} \to \mathbf{V})^\circ$. Assume $v \in (\mathbf{V} \to \mathbf{V})^\circ$, i.e. $v \equiv a_1 \Rightarrow b_1 \sqcup \ldots \sqcup a_n \Rightarrow b_n$.

$$\begin{aligned}
&\mathcal{T}(a_1 \Rightarrow b_1 \sqcup \ldots \sqcup a_n \Rightarrow b_n) \\
&= \{t \in seqs \mid \forall t_p \in elems(t) \,.\, \exists i \leq n \,.\, t_p \in \mathcal{T}(<a_i, b_i>) \wedge \\
&\qquad\qquad \forall i \leq n \,.\, \exists t_p \in elems(t) \,.\, t_p \in \mathcal{T}(<a_i, b_i>)\}
\end{aligned}$$

Applying $\mathcal{V}$ we get:

$$\begin{aligned}
&\mathcal{V}(\mathcal{T}(a_1 \Rightarrow b_1 \sqcup \ldots \sqcup a_n \Rightarrow b_n)) \\
&= \{\bot\} \cup \{f \in (\mathbf{V} \to \mathbf{V})^\circ \mid \\
&\qquad \exists t \in \mathcal{T}(f) \,. \\
&\qquad\quad \forall t_p \in elems(t) \,.\, \exists i \leq n \,.\, t_p \in \mathcal{T}(<a_i, b_i>) \wedge \\
&\qquad\quad \forall i \leq n \,.\, \exists t_p \in elems(t) \,.\, t_p \in \mathcal{T}(<a_i, b_i>)\} \\
&= \{\bot\} \cup \{f \in (\mathbf{V} \to \mathbf{V})^\circ \mid \\
&\qquad \forall <v_1, v_2> \in grf(f) \,.\, \exists i \leq n \,. \\
&\qquad\quad \exists t_p \in \mathcal{T}(<v_1, v_2>) \,.\, t_p \in \mathcal{T}(<a_i, b_i>) \wedge \\
&\qquad \forall i \leq n \,.\, \exists <v_1, v_2> \in grf(f) \,. \\
&\qquad\quad \exists t_p \in \mathcal{T}(<v_1, v_2>) \,.\, t_p \in \mathcal{T}(<a_i, b_i>)\} \\
&= \{\bot\} \cup \{f \in (\mathbf{V} \to \mathbf{V})^\circ \mid \\
&\qquad \forall <v_1, v_2> \in grf(f) \,.\, \exists i \leq n \,.\, <v_1, v_2> \in \mathcal{V}(\mathcal{T}(<a_i, b_i>)) \wedge \\
&\qquad \forall i \leq n \,.\, \exists <v_1, v_2> \in grf(f) \,.\, <v_1, v_2> \in \mathcal{V}(\mathcal{T}(<a_i, b_i>))\}
\end{aligned}$$

The last equality follows from equation (2) and the fact that

$$\{<v_1, v_2> \mid \exists t_p \in \mathcal{T}(<v_1, v_2>) \,.\, t_p \in \mathcal{T}(<a_i, b_i>)\} = \mathcal{V}(\mathcal{T}(<a_i, b_i>)) \setminus \{\bot\}$$

Assume $v' \in \mathcal{V}(\mathcal{T}(a_1 \Rightarrow b_1 \sqcup \ldots \sqcup a_n \Rightarrow b_n))$. It follows that $v' \in (\mathbf{V} \to \mathbf{V})^\circ$, i.e. $v' \equiv c_1 \Rightarrow d_1 \sqcup \ldots \sqcup c_m \Rightarrow d_m$ for some $m \geq 0$ and $c_1, \ldots, c_m, d_1, \ldots, d_m \in \mathbf{V}^\circ$. To prove $v \in \mathcal{D}^\circ[\![\sigma \to v]\!]\rho \Rightarrow v' \in \mathcal{D}^\circ[\![\sigma \to v]\!]\rho$, assume $v \in \mathcal{D}^\circ[\![\sigma \to v]\!]\rho$. We have $\forall j \leq m \,.\, \exists i \leq n \,.\, <c_j, d_j> \in \mathcal{V}(\mathcal{T}(<a_i, b_i>))$. Let $j \leq m$. Assume $c_j \in \mathcal{D}^\circ[\![\sigma]\!]\rho$. Let $i \leq n$ be such that $<c_j, d_j> \in \mathcal{V}(\mathcal{T}(<a_i, b_i>))$. By the induction hypothesis (and the equality derived in the proof case for tuples) we have $a_i \in \mathcal{D}^\circ[\![\sigma]\!]\rho$. Since $v \in \mathcal{D}^\circ[\![\sigma \to v]\!]\sigma$, $b_i \in \mathcal{D}^\circ[\![v]\!]\rho$. Then by the induction hypothesis it follows $d_i \in \mathcal{D}^\circ[\![v]\!]\rho$. This proves $v' \in \mathcal{D}^\circ[\![\sigma \to v]\!]\rho$.
The proof of the implication in the other direction, i.e. $v' \in \mathcal{D}^\circ[\![\sigma \to v]\!] \Rightarrow v \in \mathcal{D}^\circ[\![\sigma \to v]\!]$, is similar. Here, it is exploited that $\forall i \leq n \,.\, \exists j \leq m \,.\, <c_j, d_j> \in \mathcal{V}(\mathcal{T}(<a_i, b_i>))$.

This completes the proof for types not containing recursion.

Now let τ be any type containing recursion. For each recursive type $\mu t.\sigma$, we have that $\mathcal{D}^\circ[\![\mu t.\sigma]\!]\rho$ is the limit of the Cauchy sequence $<D^n(\{\bot\})>_{n \geq 0}$ where $D \equiv \lambda I \,.\, \mathcal{D}^\circ[\![\sigma]\!]\rho\{I/x\}$. Also, for all $i > 0$, $D^i(\mathcal{D}^\circ[\![v]\!]\rho) = D^{i-1}(\mathcal{D}^\circ[\![\sigma[v/t]]\!]\rho)$. Therefore, the denotation of τ may be described as the limit of a Cauchy sequence of types which do not contain the $\mu t.\sigma$ as 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. Let s be such a sequence. Assume $v \in \mathcal{D}^\circ[\![\tau]\!]\rho$. Then v is contained in an infinite numbers of types in s (Theorem 2). Since none of the types in s contain recursion, v' is contained in the same types. But then v' is also a member of τ. Thus $v' \in \mathcal{D}^\circ[\![\tau]\!]\rho$ whenever $v \in \mathcal{D}^\circ[\![\tau]\!]\rho$. The proof of the implication in the other direction is similar. □