Combining Recursive and Dynamic Types*

Hans Leiß
internet: leiss@cis.uni-muenchen.de
CIS, Universität München
Leopoldstraße 139
D-8000 München 40
Germany

Abstract. A denotational semantics of simply typed lambda calculus with a basic type `Dynamic`, modelling values whose type is to be inspected at run-time, has been given by Abadi e.a.[1]. We extend this interpretation to cover (formally contractive) recursive types as well. Soundness of typing rules and freeness of run-time type errors for well-typed programs hold.

The interpretation works also for implicitly polymorphic languages like *ML* with `Dynamic` and recursive types, and for explicitly polymorphic languages under the types-as-ideals interpretation.

1 Introduction

Static typing of programming languages has well known advantages like error detection at compile time and efficient object code free of type-checking at run-time. However, for programs that interact with storage media, other programs, or humans, it is often impossible to determine all relevant type information at compile time.

For example, consider a program operating on data that are modified after it has been compiled, are provided by a user interactively, or are fetched from external storage media. One would like the running program to inspect the type of the data, continue computation if this type is compatible with the type expected by the program, and terminate computaion or raise an exception, otherwise. Clearly, the use of such dynamic type checking should not compromise the soundness of the typing system.

In the last few years, several attempts have been made to make restricted use of dynamic typing in statically typed programming languages. A. Mycroft[13], building on ideas of M. Gordon to model dynamic typing by pairs $\langle v, \tau \rangle$ of values v with their types τ, introduced a type `Dynamic`, henceforth called `dyn`, as an infinite disjoint sum of types, each summand containing values tagged with their type. Inspection of these "dynamic values" is accomplished by a case-statement that branches according to finitely many type patterns, which exhaust the infinitely many types a dynamic value could have.

Mycroft proposed an extension of *ML*[6] by dynamic values. Actually, a version of `dyn` has been built into *CAML*[5]. The background of this implementation and alternative designs for adding dynamics to *ML* are treated by Leroy/Mauny[8]. Other

* This work has been supported by the Esprit Working Group BRA 7232, GENTZEN.

languages like L. Cardelli's[3] *AMBER* also have dynamic values. In fact, the implementation of *AMBER* took advantage of treating compiled modules as dynamic values.

M. Abadi, L. Cardelli, B. Pierce and G. Plotkin[1] investigated dynamic typing in a systematic way. They gave operational and denotational semantics of the simply typed lambda calculus with a basic type `dyn`. Additionally, they presented some ideas on polymorphically typed languages with `dyn`, focussing on difficulties in matching polymorphic types against patterns. This last aspect has been further investigated very recently by M. Abadi, L. Cardelli, B. Pierce and D. Remy[2], with extensions to languages with subtyping and abstract data types. A rather different study of dynamic type checking has been given by F. Henglein[7].

To give a denotational semantics for `dyn`, one has to establish

$$\langle v, \tau \rangle \in [\![\mathtt{dyn}]\!] \quad \text{if and only if} \quad v \in [\![\tau]\!],$$

for arbitrary values v and (closed) types τ, including $\tau = \mathtt{dyn}$. In the case of simply typed λ-calculus, Abadi e.a.[1] work in the ideal model of D. MacQueen, G. Plotkin and R. Sethi[9], and define $[\![\mathtt{dyn}]\!]$ recusively by mimicking the construction of types over the set of pairs $\langle v, \tau \rangle$. However, for the combinations of `dyn` with polymorphic languages, suggested in [1, 2] and [8], no denotational semantics was known so far.

Our main concern is to extend the model of Abadi e.a.[1] to cover recursive types $\mu\alpha.\tau$ as well, such as the trees and lists of real programming languages. When recursive types are present, the method of defining $[\![\mathtt{dyn}]\!]$ by mimicking the construction of types seems impossible: $v \in [\![\mu\alpha.\tau]\!]$ cannot be characterized by conditions of the form $u \in [\![\sigma]\!]$ for types σ simpler than τ.

A more general approach is needed in order to define the meaning of types when `dyn` and recursive types are combined. The main contribution of this paper is that while -in this situation- the recursive equations for type interpretations do *not* constitute a well-founded recursion on types, we can solve them simultaneously for all types, using Banach's fixed point theorem on an infinite product space of the metric space of ideals.

An advantage of our method is that it allows to give a denotational semantics for dynamic (and recursive) types in polymorphic languages as well. On the other hand, in this case we have neglected somewhat the semantics of terms in that only closed type-tags and first-order pattern-variables in typecase-expressions have been treated.

In Section 2.1 we present the syntax of simply typed λ-calculus with dynamic and recursive types, and in Section 2.2 review the construction of the ideal model and some preliminaries for our extension. In Section 2.3 we will construct the ideal interpretation for simply typed λ-calculus with `dyn` and recursive types. Section 3 extends this to an interpretation of dynamic and recursive types combined with explicit and implicit polymorphism.

2 Dynamic and recursive types in a simply typed language

Following ideas of M. Gordon, we view *dynamic types* as sets of pairs $\langle v, \tau \rangle$ containing a value v of type τ together with its type. Programs are allowed to access the type-tag and make their actions depend on it. Compared with sum types $(\tau_1 + \tau_2)$, whose elements $\langle v, i\rangle$ contain a tag i indicating that v is of type τ_i, dynamic types are more flexible in that their tags range over an infinite set. They can be modelled as infinite sums, with the restriction that the infinite set of tags has some algebraic structure that allows programs to use branching according to finitely many tag-*patterns*, instead of an infinite distinction on tags. A. Mycroft[12] first considered dynamic types as infinite sums of this kind.

Instead of working with different dynamic types, it is sufficient to consider one type `dyn` containing all these ⟨value,tag⟩-pairs, with the type as the algebra of tags.

2.1 Simply typed λ-calculus with `dyn` and recursive types

In this section we give the syntax of a simply typed λ-calculus with recursive types and a basic type `dyn`. It is similar to that of [1], to which we also refer for example programs. The types we consider are basic types including `dyn`, disjoint sums, product, function space, and recursive types.

Definition 1. *Types* τ and *terms* t are defined by the following grammar:

$$\tau = \alpha \mid \texttt{bool} \mid \texttt{nat} \mid \texttt{dyn} \mid (\tau + \tau) \mid (\tau \times \tau) \mid (\tau \to \tau) \mid \mu\alpha.\tau$$

$$\begin{aligned}
t = {} & x \mid \mathbf{wrong} \\
& \mid tt \mid ff \mid (\mathbf{case}\ t\ \mathbf{of}\ tt\ \mathbf{then}\ t,\ ff\ \mathbf{then}\ t) \\
& \mid 0 \mid S(t) \mid (\mathbf{case}\ t\ \mathbf{of}\ 0\ \mathbf{then}\ t,\ S(x)\ \mathbf{then}\ t) \\
& \mid (\mathbf{dynamic}\ t : \tau) \mid (\mathbf{typecase}\ t\ \mathbf{of}\ \{\alpha_1, \ldots, \alpha_n\}\ x : \tau\ \mathbf{then}\ t\ \mathbf{else}\ t) \\
& \mid in_{1,\tau+\tau}(t) \mid in_{2,\tau+\tau}(t) \mid (\mathbf{case}\ t\ \mathbf{of}\ \langle x, 1\rangle\ \mathbf{then}\ t,\ \langle x, 2\rangle\ \mathbf{then}\ t) \\
& \mid (t, t) \mid \pi_1 t \mid \pi_2 t \\
& \mid \lambda x : \tau.t \mid (t \cdot t)
\end{aligned}$$

The intended meaning of an expression (**dynamic** $t : \tau$) is to create a *dynamic value* $\langle v, \tau\rangle$, which may then be stored to external media or otherwise brought out of the control of the running program.

Conversely, the meaning of (**typecase** d **of** $\{\alpha_1, \ldots, \alpha_n\}$ $x : \tau$ **then** r **else** s) is to match the type σ of $\langle v, \sigma\rangle$, the value of d, against the pattern τ, and if the match succeeds, then continue with r –using v for x and the types found by the match for the pattern-variables α_i–, else with s. If the value of d is not a dynamic, an error occurs. (C.f. Section 2.4 and 2.5 for details.) The variables x and α_i are bound variables with scope $x : \tau$ and r.[2] To handle embedded patterns, the distinction between free and bound pattern-variables is made explicit by binding guards $\{\alpha_1, \ldots, \alpha_n\}$, as in

$$\begin{aligned}
\lambda x : \texttt{dyn}.\lambda f : \texttt{dyn}.\ (&\mathbf{typecase}\ x\ \mathbf{of}\ \{\alpha\}\ y : \alpha \times \alpha \\
&\mathbf{then}\ (\mathbf{typecase}\ f\ \mathbf{of}\ \{\beta\}\ g : \alpha \times \alpha \to \beta \\
&\qquad \mathbf{then}\ (\mathbf{dynamic}\ gy : \beta) \\
&\qquad \mathbf{else}\ (\mathbf{dynamic}\ \pi_1 y : \alpha)) \\
&\mathbf{else}\ (\mathbf{dynamic}\ \text{“not a nice pair”} : \texttt{string}))
\end{aligned}$$

[2] A more flexible syntax, (**typecase** d **of** $\{x_1, \ldots, x_m, \alpha_1, \ldots, \alpha_n\}$ $p : \tau$ **then** r **else** s) with a term-pattern p, would allow to inspect the value component as well. Essentially, the case (**typecase** d **of** $\{x_1, \ldots, x_m\}$ $p : \tau$ **then** r **else** s) is available in *CAML*.

To simplify notation, from now on we will only use typecase-expressions binding a single pattern-variable.

The meaning of the remaining expressions is fairly standard and omitted. (See also [1] for operational and denotational semantics of terms.) It may be sufficient to mention that the variable x in the case-statements is a pattern-variable bound in the corresponding then-branch.

From the terms generated by the above grammar, a subclass of *well-typed* terms is defined using a type inference system. We only give those rules of the system in Abadi e.a.[1] that deal with dynamics[3], those of [9] that deal with recursive types, and those for function types.

A *type basis* is a finite sequence Γ of *typing statements* $x : \tau$, assigning types τ to object variables x. If Γ contains several assumptions $x : \sigma$ for the same object variable x, then $\Gamma(x)$, *–the type assigned to x by Γ–* is the type σ of the rightmost of these. When writing $\Gamma \rhd t : \tau$, we always assume that Γ a typing assumption for each free variable of t.

Definition 2 (Typing Rules).

$$\text{(Var)} \qquad \frac{}{\Gamma \rhd x : \Gamma(x)}$$

$$\text{(Dyn I)}_{\mathrm{e}} \quad \frac{\Gamma \rhd t : \tau}{\Gamma \rhd (\textbf{dynamic}\ t : \tau) : \texttt{dyn}}, \quad \text{if } \tau \text{ is closed}$$

$$\text{(Dyn E)}^{-} \quad \frac{\Gamma \rhd d : \texttt{dyn}, \quad \Gamma,\ x : \tau[\rho/\alpha] \rhd r[\rho/\alpha] : \sigma \text{ for all closed } \rho, \quad \Gamma \rhd s : \sigma}{\Gamma \rhd (\textbf{typecase}\ d\ \textbf{of}\ \{\alpha\}\ x : \tau\ \textbf{then}\ r\ \textbf{else}\ s) : \sigma}$$

$$(\mu\ \text{I}) \quad \frac{\Gamma \rhd t : \mu\alpha.\tau}{\Gamma \rhd t : \tau[\mu\alpha.\tau/\alpha]} \qquad (\mu\ \text{E}) \quad \frac{\Gamma \rhd t : \tau[\mu\alpha.\tau/\alpha]}{\Gamma \rhd t : \mu\alpha.\tau}$$

$$(\rightarrow \text{I})_{\mathrm{e}} \quad \frac{\Gamma, x : \sigma \rhd t : \tau}{\Gamma \rhd \lambda x : \sigma.t : (\sigma \rightarrow \tau)} \qquad (\rightarrow \text{E}) \quad \frac{\Gamma \rhd t : \sigma \rightarrow \tau, \quad \Gamma \rhd s : \sigma}{\Gamma \rhd (t \cdot s) : \tau}$$

Note that rule (Dyn E)^{-} has an infinite number of premisses; these capture what is needed in the soundness Theorem 18. For practical purposes, it is more natural to use the following more restrictive rule:

$$\text{(Dyn E)} \quad \frac{\Gamma \rhd d : \texttt{dyn}, \quad \Gamma,\ x : \tau \rhd r : \sigma, \quad \Gamma \rhd s : \sigma}{\Gamma \rhd (\textbf{typecase}\ d\ \textbf{of}\ \{\alpha\}\ x : \tau\ \textbf{then}\ r\ \textbf{else}\ s) : \sigma}, \quad \alpha \text{ not free in } \Gamma, \sigma.$$

This demands that there is a uniform proof of $\Gamma,\ x : \tau[\rho/\alpha] \rhd r[\rho/\alpha] : \sigma$ for all ρ.

Example 1. Assuming a further base type `Unit` with single element $()$, one can define the type of lists of naturals as the recusive type $\texttt{nat-list} := \mu\alpha.(\texttt{Unit} + (\texttt{nat} \times \alpha))$. From assumptions

$$\begin{aligned} \Gamma_{\texttt{nat}} = () : \texttt{Unit}, \mathrm{in}_1 : \texttt{Unit} \rightarrow (\texttt{Unit} + (\texttt{nat} \times \texttt{nat-list})), \\ \mathrm{in}_2 : \texttt{nat} \times \texttt{nat-list} \rightarrow (\texttt{Unit} + (\texttt{nat} \times \texttt{nat-list})) \end{aligned}$$

[3] The added restriction on closed types in $\text{(Dyn I)}_{\mathrm{e}}$ is needed to define the semantics for types as in Theorem 13.

we can derive typings like $\Gamma_{\texttt{nat}} \rhd [0,1,3] : \texttt{nat-list}$, using $[] := \mathrm{in}_1()$, $[n,x] := \mathrm{in}_2(n,x)$, $1 := S(0)$ etc. Indirectly, we can also define a type of inhomogeneous lists as $\texttt{dyn-list} := \mu\alpha.(\texttt{Unit} + (\texttt{dyn} \times \alpha))$. With corresponding typing basis, one can derive type `dyn-list` for the list

$$[(\textbf{dynamic}\ tt : \texttt{bool}), (\textbf{dynamic}\ \lambda x : \texttt{nat}.S(S(x)) : \texttt{nat} \to \texttt{nat})].$$

2.2 Preliminaries for the ideal model

A *domain* is a complete partial order $(D, \leq, \perp)$ with least element $\perp$ such that (i) every bounded subset $X \subseteq D$ has a least upper bound, $\bigsqcup X \in D$, (ii) D has only countably many finite elements, and (iii) for any $d \in D$, $\{e \mid e \text{ finite},\ e \leq d\}$ is directed and d is its least upper bound. An element $d \in D$ is *finite*, if for all directed $X \subseteq D$ with $d \leq \bigsqcup X$ there is some $x \in X$ such that $d \leq x$.

In the following we will work in a domain $\mathcal{V}$ satisfying the recursion equation

$$V \cong D_{\texttt{bool}} + D_{\texttt{nat}} + (V+V) + (V \times V) + (V \to V) + (V * \mathit{clType}) + \{\mathit{error}\}_\perp, \quad (1)$$

where $D_{\texttt{bool}}$ and $D_{\texttt{nat}}$ are the flat domains of the booleans and natural numbers, $\{\mathit{error}\}_\perp$ the flat domain of an element *error* modelling run-time errors of programs, $+$ denotes the disjoint sum, $\times$ the cartesian product and $\to$ the space of continuous functions of two domains. Finally, $*$ constructs from a domain D and a set A the domain with universe

$$\{\langle v, a\rangle \mid v \in D - \{\perp\},\ a \in A\} \cup \{\perp\}$$

and the natural partial ordering inherited from D. If D is one of the summands of V, we write d^V for the injection of $d \in D$ into V, and use

$$v\lceil_D = \begin{cases} d, & \text{if } v = d^V \text{ and } d \in D \\ \perp_D, & \text{else} \end{cases}$$

Definition 3. Let $\mathcal{V} = (V, \leq, \perp)$ be a complete partial order. $I \subseteq V$ is an *ideal* of V, iff (i) $\perp_V \in I$, (ii) the supremum of every directed subset of I belongs to I, and (iii) I is downward closed, i.e. $a \leq b \in I$ implies $a \in I$. Let $\mathcal{I}$ be the set of all ideals of $\mathcal{V}$.

Definition 4. Let I and J be ideals of $\mathcal{V}$, and τ a single and A a set of closed types. Define the following subsets of V:

$$\begin{aligned}
(I + J) &:= \{\langle i, 1\rangle^V \mid i \in I - \{\perp_V\}\} \cup \{\langle j, 2\rangle^V \mid j \in J - \{\perp_V\}\} \cup \{\perp_V\} \\
(I \times J) &:= \{\langle i, j\rangle^V \mid i \in I,\ j \in J\} \cup \{\perp_V\} \\
(I \to J) &:= \{f^V \mid f \in (V \to V),\ f(I) \subseteq J\} \cup \{\perp_V\} \\
I * A &:= \{\langle i, \tau\rangle^V \mid i \in I,\ \tau \in A\} \cup \{\perp_V\}
\end{aligned}$$

These subsets are partially ordered as follows. Take $\perp_V$ as least element, compare pairs $\langle v, \tau\rangle^V$ of $I * A$ with the same type component according to their value component, compare f^V's of $(I \to J)$ according to the ordering on $V \to V$, and on $(I \times J)$ let $\langle i, j\rangle^V \leq \langle k, l\rangle^V$ be true iff $i \leq k$ and $j \leq l$ on I and J. On $I + J$, elements with the same tag are compared as their value components are on I or J.

Proposition 5. *([1])* $(I + J)$, $(I \times J)$, $(I \to J)$ *and* $I * A$ *are ideals.*

Definition 6. From now on we assume that the domain solution $\mathcal{V}$ of equation (1) we are working in is the limit of domains

$$\begin{aligned} V_0 &= \{\bot_V\} \\ V_{n+1} &= D_{\texttt{bool}} + D_{\texttt{nat}} + (V_n + V_n) + (V_n \times V_n) \\ &\quad + (V_n \to V_n) + (V_n * clType) + \{error\}_\bot. \end{aligned}$$

The *rank* $rk(\mathrm{v})$ of $v \in V$ is the least $n \in \omega$ such that $v \in V_n$. The *distance* $d(I, J)$ of ideals $I, J \in \mathcal{I}$ is

$$d(I,J) = \begin{cases} 0, & \text{if } I = J \\ 2^{-min\{rk(v) \mid v \in I \bowtie J\}}, & \text{if } \emptyset \neq I \bowtie J \end{cases}$$

where $I \bowtie J := \{v \mid v \text{ a finite element of } I - J \text{ or } J - I\}$.

Lemma 7. *([1])* $(\mathcal{I}, d)$ *is a complete metric space. In fact,* d *is an ultrametric, i.e. satisfies* $d(I,J) \leq max\{d(I,K), d(K,J)\}$ *rather than just the triangle inequality of a metric.*

A function $f : (X, d_X) \to (Y, d_Y)$ between metric spaces is *c-contractive* (resp. *non-expansive*), if $d_Y(f(x_1), f(x_2)) \leq c \cdot d_X(x_1, x_2)$ for all $x_1, x_2 \in X$, where $0 \leq c < 1$ (resp. $0 \leq c \leq 1$). Recall that by Banach's theorem, every c-contractive mapping $f : (X, d_X) \to (X, d_X)$ on a complete metric space has a unique fixed point, $\mathit{fix}\, x.\, f(x)$. Moreover, we will need the following facts:

Lemma 8. *Let* $(X_j, d_j)_{j \in J}$ *be a family of complete ultra-metric spaces, such that* $d_j(x, y) \leq 1$ *for all* $j \in J$ *and* $x, y \in X_j$. *Let* $(X, d) := (\Pi_{j \in J} X_j, \sup_{j \in J} d_j)$ *be their cartesian product, equipped with* $d(< x_j >_{j \in J}, < y_j >_{j \in J}) := sup\{d_j(x_j, y_j) \mid j \in J\} \leq 1$

1. (X, d) *is a complete ultra-metric space.*
2. *If* $\{f_j : X_j \to X \mid j \in J\}$ *a family of c-contractive mappings (for fixed c), then* $f : X \to X$, *defined by* $f(< x_j >_{j \in J}) :=< f_j(x_j) >_{j \in J}$, *is c-contractive.*
3. *If, for some* $c < 1$, f *is c-contractive in* x_k *when keeping the others components* x_j *fixed, then* $\lambda x \in X.\ \mathit{fix}\ x_k.\ < f_j(x_j) >_{j \in J}$ *is c-contractive.*
4. $h \circ g)$ *is contractive if* g *is contractive and* h *is non-expanding, or vice versa.*

Proposition 9. *([1]) On* $(\mathcal{I}, d)$, $+$, $\times$, $\to$ *and* $*\{\tau\}$ *are* $1/2$*-contractive functions.*

The intersection of an arbitrary nonempty set $\mathcal{J}$ of ideals is an ideal. Since its union in general is not, one has to consider $\bigsqcup \mathcal{J} := \bigcap\{I \in \mathcal{I} \mid \bigcup \mathcal{J} \subset I\}$.

Proposition 10. *Let* $\{I_k \mid k \in K\}$ *and* $\{J_k \mid k \in K\}$ *be nonempty families of ideals. Then (i)* $d(\bigcap_{k \in K} I_k, \bigcap_{k \in K} J_k) \leq sup_{k \in K} d(I_k, J_k)$ *and (ii)* $d(\bigsqcup_{k \in K} I_k, \bigsqcup_{k \in K} J_k) \leq sup_{k \in K} d(I_k, J_k)$.

Proof. (i) Since $Y := \bigcap\{I_k \mid k \in K\} \bowtie \bigcap\{J_k \mid k \in K\} \subseteq \bigcup_{k \in K}(I_k \bowtie J_k)$, we have $2^{min\ rk(Y)} \geq 2^{min_{k \in K} min\ rk(I_k \bowtie J_k)} = min_{k \in K} 2^{min\ rk(I_k \bowtie J_k)}$, which implies the claim. (ii) Similarly, we use that $X := \bigsqcup\{I_k \mid k \in K\} \bowtie \bigsqcup\{J_k \mid k \in K\} \subseteq \bigcup_{k \in K}(I_k \bowtie J_k)$. For this, note that a finite element of $\bigsqcup\{I_k \mid k \in K\}$ already belongs to some I_k.

2.3 Semantics of types as ideals

Following MacQueen e.a.[9], we use Banach's fixed point theorem to define the meaning $[\![\mu\alpha.\tau]\!]\eta$ of a recursive type $\mu\alpha.\tau$ as the fixed-point of $\lambda J \in \mathcal{I}.[\![\tau]\!]\eta[J/\alpha]$. Since only contractive mappings are guaranteed to have (unique) fixed points, we have to restrict ourselves to a subclass of all types.

Definition 11. ([9]) *Well-formed types* are the following subclass of types:

$$\begin{aligned}\tau = \alpha \mid \texttt{bool} \mid \texttt{nat} \mid \texttt{dyn} \mid (\tau+\tau) \mid (\tau\times\tau) \mid (\tau\to\tau) \mid \\ \mid \mu\alpha.\tau, \quad \text{provided } \tau \text{ is formally contractive in } \alpha.\end{aligned}$$

A type τ is *formally contractive in* α, if either (i) τ is atomic and α is not free in τ, or (ii) τ is of the form $(\tau_1+\tau_2)$, $(\tau_1\times\tau_2)$ or $(\tau_1\to\tau_2)$, or (iii) τ is of the form $\mu\beta.\sigma$ and σ is contractive in α, or $\alpha\equiv\beta$.

From now on, "type" means "well-formed type". We write *clType* for the set of *closed* types, containing no free variables, and *Type* for the set of all types.

Well-formed types are those not containing a subtype of the form $\mu\alpha_1\ldots\mu\alpha_n.\alpha_i$, $1\le i\le n$. They form a rich class of type expressions which define contractive mappings on $(\mathcal{I},d)$.

According to the informal discussion of dynamic values in Section 1, the type `dyn` should be interpreted by the collection of all pairs $\langle v,\tau\rangle^V$ where $v\in[\![\tau]\!]$ and $\tau\in \mathit{clType}$. Abadi e.a.[1] give such an interpretation by mimicking the ordinary type constructors $+$, $\times$ and $\to$ by new "dynamic" constructors $\dot{+}$, $\dot{\times}$, and $\dot{\to}$ on the universe $V * \mathit{clType}$ of dynamic values. These constructors combine *dynamic* types $I,J\subseteq (V*\mathit{clType})^V$ as follows (but also work for arbitrary ideals):

$$\begin{aligned}(I\dot{+}J) &:= \{\langle\langle i,1\rangle^V,\sigma+\tau\rangle^V \mid \langle i,\sigma\rangle^V\in I, \tau\in \mathit{clType}\}\\ &\quad\cup\{\langle\langle j,2\rangle^V,\sigma+\tau\rangle^V \mid \sigma\in\mathit{clType}, \langle j,\tau\rangle^V\in J\} \cup \{\bot_V\}.\\ (I\dot{\times}J) &:= \{\langle\langle i,j\rangle^V,\sigma\times\tau\rangle^V \mid \langle i,\sigma\rangle^V\in I, \langle j,\tau\rangle^V\in J\} \cup \{\bot_V\}.\\ (I\dot{\to}J) &:= \{\langle f^V,\sigma\to\tau\rangle^V \mid f\in(V\to V),\\ &\qquad \langle f(i),\tau\rangle^V\in J \text{ for all } i\in V \text{ with } \langle i,\sigma\rangle^V\in I\}\cup\{\bot_V\},\end{aligned}$$

Since these fucntions are contractive on $\mathcal{I}$, $[\![\texttt{dyn}]\!]$ can recursively be defined by

$$\begin{aligned}[\![\texttt{dyn}]\!] = [\![\texttt{bool}]\!]*\{\texttt{bool}\} \cup [\![\texttt{nat}]\!]*\{\texttt{nat}\} \cup [\![\texttt{dyn}]\!]*\{\texttt{dyn}\}\\ \cup([\![\texttt{dyn}]\!]\dot{+}[\![\texttt{dyn}]\!]) \cup ([\![\texttt{dyn}]\!]\dot{\times}[\![\texttt{dyn}]\!]) \cup ([\![\texttt{dyn}]\!]\dot{\to}[\![\texttt{dyn}]\!]).\end{aligned}$$

In the presence of recursive types, however, this method seems no longer be applicable, as it relies on the fact that for non-basic τ, $\langle v,\tau\rangle^V\in[\![\texttt{dyn}]\!]$ is characterized by elements $\langle u,\sigma\rangle^V\in[\![\texttt{dyn}]\!]$ with *simpler* types σ. This is not the case when choosing

$$\dot{\mu}(I) := \{\langle v,\mu\alpha.\tau\rangle^V \mid \langle v,\tau[\mu\alpha.\tau/\alpha]\rangle^V\in I\} \cup \{\bot_V\}.$$

Also, $d(\dot{\mu}([\![\texttt{bool}]\!]*\{\texttt{bool}\}),\dot{\mu}([\![\texttt{nat}]\!]*\{\texttt{nat}\})) = d([\![\texttt{bool}]\!]*\{\texttt{bool}\},[\![\texttt{nat}]\!]*\{\texttt{nat}\})$ shows that we would not get a contractive mapping.

So we take a more general approach to define $[\![\texttt{dyn}]\!]$ that is fairly independent of the choice of type constructors, and formalizes the intuitive meaning directly.

Definition 12. Let *Env* be the set of all assignments $\eta : TypeVar \to \mathcal{I}$. A *meaning of types* as ideals in $\mathcal{V}$ is a function $[\![\cdot]\!]\cdot$, such that the following conditions hold for all types τ and assignments $\eta \in Env$:

$$
\begin{array}{llll}
[\![\alpha]\!]\eta & = \eta(\alpha) & [\![(\tau_1 + \tau_2)]\!]\eta & = ([\![\tau_1]\!]\eta + [\![\tau_2]\!]\eta) \\
[\![\mathtt{bool}]\!]\eta & = D_{\mathtt{bool}} \cup \{\bot_V\} & [\![(\tau_1 \times \tau_2)]\!]\eta & = ([\![\tau_1]\!]\eta \times [\![\tau_2]\!]\eta) \\
[\![\mathtt{nat}]\!]\eta & = D_{\mathtt{nat}} \cup \{\bot_V\} & [\![(\tau_1 \to \tau_2)]\!]\eta & = ([\![\tau_1]\!]\eta \to [\![\tau_2]\!]\eta) \\
[\![\mathtt{dyn}]\!]\eta & = \bigcup_{\tau \in clType} [\![\tau]\!]\eta * \{\tau\} & [\![\mu\alpha.\tau]\!]\eta & = [\![\tau]\!]\eta[[\![\mu\alpha.\tau]\!]\eta/\alpha]
\end{array}
$$

Without the clause for `dyn`, it is easily seen by induction on τ that $[\![\tau]\!]\eta$ exists for all η, once it is clear that the fixed point of $\lambda J \in \mathcal{I}.[\![\tau]\!]\eta[I/\alpha]$ is an ideal of $\mathcal{V}$. However, with the clause for `dyn` we can no longer apply induction on τ to ensure the existence of $[\![\tau]\!]\eta$: the above equations specify $[\![\mathtt{dyn}]\!]\eta$ using the values of arbitrarily complicated types - possibly containing the type `dyn`.

The main point of this paper is to show that, in spite of the apparently non-well-founded recursion in the clauses of Definition 12, $[\![\tau]\!]\eta$ makes perfect sense.[4] An infinite simultaneous recursion and the uniqueness of fixed-points in complete metric spaces is used to ensure that $[\![\cdot]\!]\cdot$ exists.

Theorem 13. *There is a mapping* $[\![\cdot]\!]\cdot : Type \times Env \to \mathcal{I}$ *satisfying the conditions of Definition 12, i.e.* $[\![\tau]\!]\eta$ *is well-defined for all* τ *and* η.

Proof. Let $\Pi\mathcal{I} = \Pi_{\tau \in Type}\, \mathcal{I}_\tau$, with $\mathcal{I}_\tau = \mathcal{I}$, be the product space of the space of ideals of $\mathcal{V}$, indexed by all types. Define $F = \langle F_\tau \rangle_{\tau \in Type} : \Pi\mathcal{I} \times Env \to \mathcal{I}$ as follows, where $I \in \Pi\mathcal{I}$ is written as a function $I(\tau)$:

$$
\begin{array}{llll}
F_\alpha(I,\eta) & = \eta(\alpha) & F_{(\tau_1+\tau_2)}(I,\eta) & = F_{\tau_1}(I,\eta) + F_{\tau_2}(I,\eta) \\
F_{\mathtt{bool}}(I,\eta) & = D_{\mathtt{bool}} \cup \{\bot_V\} & F_{(\tau_1\times\tau_2)}(I,\eta) & = F_{\tau_1}(I,\eta) \times F_{\tau_2}(I,\eta) \\
F_{\mathtt{nat}}(I,\eta) & = D_{\mathtt{nat}} \cup \{\bot_V\} & F_{(\tau_1\to\tau_2)}(I,\eta) & = F_{\tau_1}(I,\eta) \to F_{\tau_2}(I,\eta) \\
F_{\mathtt{dyn}}(I,\eta) & = \bigcup\{I(\tau) * \{\tau\} \mid \tau \in clType\} & F_{\mu\alpha.\tau}(I,\eta) & = \mathit{fix}\, J \in \mathcal{I}.\, F_\tau(I,\eta[J/\alpha]).
\end{array}
$$

Claim 1. For each type τ and $\eta \in Env$, $d(F_\tau(I,\eta), F_\tau(I',\eta)) \le 1/2 \cdot\ d(I,I')$.

Proof by induction on τ:

$\tau \in \{\alpha,\ \mathtt{bool},\ \mathtt{nat}\}$: Then $d(F_\tau(I,\eta), F_\tau(I',\eta)) = 0 \le 1/2 \cdot d(I,I')$.

$\tau = \mathtt{dyn}$: First note that $F_{\mathtt{dyn}}(I,\eta) \in \mathcal{I}$, because if any two ideals of an arbitrary union of ideals have incomparable non-bottom elements only, then this union is an ideal. Next, we have

$$
\begin{array}{ll}
d(F_{\mathtt{dyn}}(I,\eta), F_{\mathtt{dyn}}(I',\eta) & \\
\le sup\{d(I(\tau) * \{\tau\}, I'(\tau) * \{\tau\}) \mid \tau \in clType\} & \text{(by Proposition 10)} \\
\le 1/2 \cdot sup\{d(I(\tau), I'(\tau)) \mid \tau \in clType\} & \\
\le 1/2 \cdot d(I,I'). &
\end{array}
$$

---

[4] Without recursive types, we might add a clause for variables like

$$[\![\mathrm{dyn}]\!]\eta = \bigcup\{[\![\alpha]\!]\eta * \{\alpha\} \mid \alpha \text{ a type variable}\} \ \cup\ \ldots$$

to the definition and still show that $[\![\tau]\!]\eta$ exists. But then if τ contains `dyn`, $[\![\tau]\!]\eta$ depends on *all* the $[\![\alpha]\!]\eta$ and hence the substitution lemma below fails for such τ.

$\tau = (\tau_1 \circ \tau_2)$, where $\circ$ is one of $+$, $\times$, or $\rightarrow$: We can use Proposition 9.

$\tau = \mu\alpha.\sigma$: By induction, we have $d(F_\sigma(I,\theta), F_\sigma(I',\theta)) \leq 1/2 \cdot d(I,I')$, for each θ, in particular for each $\theta = \eta[J/\alpha]$, where $J \in \mathcal{I}$. It is sufficient to show the following claim, whose proof by induction on σ is standard.

Claim 2. Suppose σ is formally contractive in $\alpha_1, \ldots, \alpha_n$. The map $(I, J_1, \ldots, J_n) \mapsto F_\sigma(I, \eta[J_1/\alpha_1, \ldots, J_n/\alpha_n])$ is 1/2-contractive.

Using Claim 2 and Lemma 8, we get that $I \mapsto \mathit{fix}\, J.\, F_\sigma(I, \eta[J/\alpha])$ is 1/2-contractive, and hence

$$\begin{aligned} d(F_{\mu\alpha.\sigma}(I,\eta), F_{\mu\alpha.\sigma}(I',\eta)) &= d(\mathit{fix}\, J.\, F_\sigma(I,\eta[J/\alpha]), \mathit{fix} J.\, F_\sigma(I',\eta[J/\alpha])) \\ &\leq 1/2 \cdot d(I,I'), \end{aligned}$$

which finishes the proof of Claim 1.

From Claim 1 we conclude that on the product space $\Pi\mathcal{I}$, $d(F(I,\eta), F(I',\eta)) \leq 1/2 \cdot\ d(I,I')$, and so F is contractive, for fixed η. By Banach's theorem, for each η there is a (unique) element $I_\eta \in \Pi\mathcal{I}$ such that $F(I_\eta, \eta) = I_\eta$. We now define

$$[\![\tau]\!]\eta := I_\eta(\tau) = F_\tau(I_\eta, \eta).$$

Note that induction on types cannot be used to show that $[\![\cdot]\!]\cdot$ satisfies the conditions of Definition 12.

Claim 3. $F_\tau(I,\rho)$ only depends on $\rho(\alpha)$, with α free in τ, and on $I(\sigma)$ for closed σ.

This is easily seen by induction on τ, since the case for $\tau = \mathtt{dyn}$ is obvious. Next we use the uniqueness of fixed points to show:

Claim 4. If $I_\eta = F(I_\eta, \eta)$ and $I_\theta = F(I_\theta, \theta)$, then $I_\eta(\sigma) = I_\theta(\sigma)$ for all *closed* σ.

Proof: By Claim 3, $I_\eta(\sigma) = F_\sigma(I_\eta, \eta)$ depends only on all the $I_\eta(\sigma')$ for closed σ'. Note that this dependency is contractive, so

$$d(I_\eta(\sigma), I_\theta(\sigma)) \leq 1/2 \cdot \mathit{sup}\{d(I_\eta(\sigma'), I_\theta(\sigma')) \mid \sigma' \in \mathit{clType}\}.$$

Since this holds for all closed types, the right hand side must be 0.

Claim 5. $[\![\cdot]\!]\cdot$ satisfies the conditions of Definition 12.

Proof: This is obvious for all types except the recursive ones. For these, use

$$\begin{aligned} [\![\mu\alpha.\tau]\!]\eta &= F_{\mu\alpha.\tau}(I_\eta, \eta) \\ &= \mathit{fix}\, J \in \mathcal{I}.\, F_\tau(I_\eta, \eta[J/\alpha]) \\ &= \mathit{fix}\, J \in \mathcal{I}.\, F_\tau(I_{\eta[J/\alpha]}, \eta[J/\alpha]) \qquad \text{(by Claims 3 and 4)} \\ &= \mathit{fix}\, J \in \mathcal{I}.\, [\![\tau]\!]\eta[J/\alpha] \\ &= [\![\tau]\!]\eta[[\![\mu\alpha.\tau]\!]\eta/\alpha]. \end{aligned}$$

Corollary 14. *(i)* $[\![\tau]\!]\eta$ *does not depend on* $\eta(\alpha)$ *for* α *not free in* τ. *(ii) If* τ *is contractive in* α, *then* $\lambda J \in \mathcal{I}.\ [\![\tau]\!]\eta[J/\alpha]$ *is 1/2-contractive.*

Corollary 15. *(Substitution Lemma)* $[\![\sigma]\!]\eta[[\![\tau]\!]\eta/\alpha] = [\![\sigma[\tau/\alpha]]\!]\eta$.

Proof. By induction on σ. The claim is obvious if σ is a type variable, and immediate by Proposition 9 if σ is $(\sigma_1 + \sigma_2)$, $(\sigma_1 \times \sigma_2)$, or $(\sigma_1 \to \sigma_2)$. Let θ be $\eta[[\![\tau]\!]\eta/\alpha]$.
$\sigma \in \{\mathtt{bool}, \mathtt{nat}, \mathtt{dyn}\}$: $[\![\sigma]\!]\theta = I_\theta(\sigma) = I_\eta(\sigma) = [\![\sigma]\!]\eta = [\![\sigma[\tau/\alpha]]\!]\eta$, using Claim 4.
$\sigma = \mu\beta.\rho$: We may assume $\alpha \not\equiv \beta$ and $\mathit{free}(\tau) \cap \mathit{bound}(\sigma) = \emptyset$, and hence

$$\begin{aligned}
[\![\sigma]\!]\theta = [\![\mu\beta.\rho]\!]\theta &= \mathit{fix}\, J \in \mathcal{I}.\, [\![\rho]\!]\theta[J/\beta] \\
&= \mathit{fix}\, J \in \mathcal{I}.\, [\![\rho]\!]\eta[J/\beta][[\![\tau]\!]\eta/\alpha] && \text{(by disjointness of variables)} \\
&= \mathit{fix}\, J \in \mathcal{I}.\, [\![\rho[\tau/\alpha]]\!]\eta[J/\beta] && \text{(by induction and Cor. 14 (i))} \\
&= [\![\mu\beta.\rho[\tau/\alpha]]\!]\eta \\
&= [\![(\mu\beta.\rho)[\tau/\alpha]]\!]\eta = [\![\sigma[\tau/\alpha]]\!]\eta && \text{(by disjointness of variables).}
\end{aligned}$$

Immediate consequences are theorems 18 and 19 of the following sections, extending those of Abadi e.a.[1] for the corresponding type system without recursive types.

2.4 Soundness of typing rules

The meaning of terms is defined along Milner's[11] original description of the ideal model. We only give the clauses for **wrong** and dynamics:

Definition 16. Let $\mathit{match}_{\{\alpha_1,\ldots,\alpha_n\}}(\sigma, \tau) = S$ say that $S : \{\alpha_1, \ldots, \alpha_n\} \to \mathit{Type}$ is a substitution such that $\sigma \equiv \tau S$.

$[\![\mathbf{wrong}]\!]\eta := \mathit{error}^V$

$$[\![(\mathbf{dynamic}\ t : \tau)]\!]\eta := \begin{cases} \mathit{error}^V, & \text{if } [\![t]\!]\eta = \mathit{error}^V \\ \langle [\![t]\!]\eta, \tau\rangle^V, & \text{otherwise} \end{cases}$$

$[\![(\mathbf{typecase}\ d\ \mathbf{of}\ \{\alpha\}\ x : \tau\ \mathbf{then}\ r\ \mathbf{else}\ s)]\!]\eta :=$

$$\begin{cases} [\![r[\rho/\alpha]]\!]\eta[v/x], & \text{if } \langle v, \sigma\rangle = [\![d]\!]\eta\lceil_{(V*clType)} \text{ and } \mathit{match}_{\{\alpha\}}(\sigma,\tau) = [\rho/\alpha] \neq \mathit{fail} \\ [\![s]\!]\eta, & \text{if } \langle v, \sigma\rangle = [\![d]\!]\eta\lceil_{(V*clType)} \text{ and } \mathit{match}_{\{\alpha\}}(\sigma,\tau) = \mathit{fail}, \\ & \quad \text{or } [\![d]\!]\eta = \bot_V \\ \mathit{error}^V, & \text{else} \end{cases}$$

The typing rules (including the familiar ones not given) can be shown to be sound with respect to the denotational meanings of types and terms. We have to restrict type assignments to have values in the set of *semantic types* of $\mathcal{V}$,

$$\mathcal{T} := \{I \in \mathcal{I} \mid \mathit{error}^V \notin I\}.$$

The typing rules are chosen such that **wrong** is untypable. It is easily seen that *no* type contains error^V:

Lemma 17. *If $\eta : \mathit{TypeVar} \to \mathcal{T}$, then $[\![\tau]\!]\eta \in \mathcal{T}$ for each type τ.*

Note that we cannot use induction on types to prove Lemma 17. Instead, observe that in the metric space $(\mathcal{I}, d)$, a Cauchy sequence contained in $\mathcal{T}$ never converges to an ideal $I \notin \mathcal{T}$.

The following theorem, which can be shown by induction on the proof of $\Gamma \triangleright s : \sigma$, implies that no typable term denotes error^V.

Theorem 18. *(c.f. [1]) Suppose $\eta(x) \in \Gamma(x) \in \mathcal{T}$ whenever $\Gamma(x)$ is defined. If $\Gamma \triangleright t : \tau$ is provable, then $[\![t]\!]\eta \in [\![\tau]\!]\eta$.*

2.5 Soundness of evaluation

There is an operational notion of evaluation, as defined in Abadi e.a.[1], which is correct with respect to the denotational one. Only closed expressions are evaluated, and the result is a term in canonical form. *Terms in canonical form*, or (operational) *values* v, are given by the grammar

$$\begin{array}{lll}
v = \mathbf{wrong} \mid u & & (\textit{values}) \\
u = \mathit{tt} \mid \mathit{ff} \mid n & & (\textit{proper values}) \\
\quad\mid (u, u) & & \\
\quad\mid in_{1,\tau+\sigma}\ u \mid in_{2,\tau+\sigma}\ u & & \\
\quad\mid \lambda x : \tau . t, \qquad \textit{if } \tau \textit{ is closed and free}(t) \subseteq \{x\} & & \\
\quad\mid (\mathbf{dynamic}\ u : \tau), \qquad \textit{if } \tau \textit{ is closed} & & \\
n = 0 \mid S(n) & & (\textit{natural values}).
\end{array}$$

Inductively, it is defined when *closed term t reduces to canonical form v*, written as $t \Rightarrow v$. Again, we only give the rules for expressions dealing with **wrong** and dynamics (with u and v as above):

$$(\Rightarrow_{\mathtt{wrong}})\ \frac{}{\mathbf{wrong} \Rightarrow \mathbf{wrong}}$$

$$(\Rightarrow_{\mathtt{dyn},1})\ \frac{t \Rightarrow u}{(\mathbf{dynamic}\ t : \tau) \Rightarrow (\mathbf{dynamic}\ u : \tau)}$$

$$(\Rightarrow_{\mathtt{dyn},2})\ \frac{t \Rightarrow \mathbf{wrong}}{(\mathbf{dynamic}\ t : \tau) \Rightarrow \mathbf{wrong}}$$

$$(\Rightarrow_{\mathtt{tc},1})\ \frac{d \Rightarrow (\mathbf{dynamic}\ u : \sigma), \quad r[\rho/\alpha][u/x] \Rightarrow v}{(\mathbf{typecase}\ d\ \mathbf{of}\ \{\alpha\}\ x : \tau\ \mathbf{then}\ r\ \mathbf{else}\ s) \Rightarrow v},\ match_{\{\alpha\}}(\sigma, \tau) = [\rho/\alpha]$$

$$(\Rightarrow_{\mathtt{tc},2})\ \frac{d \Rightarrow (\mathbf{dynamic}\ u : \sigma), \qquad s \Rightarrow v}{(\mathbf{typecase}\ d\ \mathbf{of}\ \{\alpha\}\ x : \tau\ \mathbf{then}\ r\ \mathbf{else}\ s) \Rightarrow v},\ match_{\{\alpha\}}(\sigma, \tau) = \mathit{fail}$$

$$(\Rightarrow_{\mathtt{tc},3})\ \frac{d \Rightarrow v \qquad v \neq (\mathbf{dynamic}\ u : \sigma)}{(\mathbf{typecase}\ d\ \mathbf{of}\ \{\alpha\}\ x : \tau\ \mathbf{then}\ r\ \mathbf{else}\ s) \Rightarrow \mathbf{wrong}},$$

Next one can show that operational evaluation preserves types and denotational value. Together with the results of the previous section, this ensures that well-typed expressions t "do not cause run-time errors", i.e. $t \Rightarrow \mathbf{wrong}$ is impossible.

Theorem 19. *(c.f. [1]) Let t be closed with respect to object- and typevariables. If $t \Rightarrow v$, then (a) $[\![t]\!] = [\![v]\!]$ and (b) if $\rhd\ t : \sigma$ is provable, so is $\rhd\ v : \sigma$.*

3 Dynamic and recursive types in polymorphic languages

We can extend the combination of recursive and dynamic types from simply to polymorphically typed λ-calculus.

3.1 Explicit polymorphism

The set of types for explicit polymorphism is given by the grammar

$$\tau = \alpha \mid \texttt{bool} \mid \texttt{nat} \mid \texttt{dyn} \mid (\tau + \tau) \mid (\tau \times \tau) \mid (\tau \to \tau) \mid \mu\alpha.\tau \mid \forall\alpha.\tau \mid \exists\alpha.\tau.$$

The intended meaning of type quantifiers in the ideal model is given by

$$[\![\forall\alpha.\tau]\!]\eta = \bigcap_{J \in \mathcal{T}} [\![\tau]\!]\eta[J/\alpha] \qquad \text{and} \qquad [\![\exists\alpha.\tau]\!]\eta = \bigsqcup_{J \in \mathcal{T}} [\![\tau]\!]\eta[J/\alpha], \tag{2}$$

where $\mathcal{T}$ is the set of ideals of $\mathcal{V}$ that do not contain $error^V$. Define $\forall\beta.\tau$ and $\exists\beta.\tau$ to be formally contractive in α just as for $\mu\beta.\tau$ in Section 2.3, and let $\forall\beta.\tau$ and $\exists\beta.\tau$ be well-formed if τ is. Restricting to well-formed types, we obtain:

Theorem 20. *There is a meaning function $[\![\tau]\!]\eta$ for the ideal interpretation of polymorphic types satisfying the conditions of Definition 12 and the equations (2).*

Proof. We modify the function F from the proof of Theorem 13 by adding component functions $F_{\forall\alpha.\tau}$ and $F_{\exists\alpha.\tau}$ defined by

$$F_{\forall\alpha.\tau}(I, \eta) := \bigcap_{J \in \mathcal{T}} F_\tau(J, \eta[J/\alpha]), \quad \text{and} \quad F_{\exists\alpha.\tau}(I, \eta) := \bigsqcup_{J \in \mathcal{T}} F_\tau(J, \eta[J/\alpha]).$$

The proof of Theorem 13 carries over, once we have shown:

Claim 6. $F_{\forall\alpha.\tau}$ and $F_{\exists\alpha.\tau}$ are 1/2-contractive in their first arguments.

For $F_{\exists\alpha.\tau}$, the proof is similar to the one for $F_{\forall\alpha.\tau}$:

$$\begin{aligned}
& d(F_{\forall\alpha.\tau}(I, \eta), F_{\forall\alpha.\tau}(I', \eta)) \\
& = d(\textstyle\bigcap_{J \in \mathcal{T}} F_\tau(I, \eta[J/\alpha]), \bigcap_{J \in \mathcal{T}} F_\tau(I', \eta[J/\alpha])) \\
& \le sup_{J \in \mathcal{T}}\ d(F_\tau(I, \eta[J/\alpha]), F_\tau(I', \eta[J/\alpha])) && \text{(by Proposition 10)} \\
& \le sup_{J \in \mathcal{T}}\ 1/2 \cdot d(I, I') = 1/2 \cdot d(I, I') && \text{(by induction).}
\end{aligned}$$

The following is shown exactly as for the case $\mu\beta.\rho$ in Proposition 15.

Proposition 21. *(Substitution Lemma)*

$$[\![\forall\beta.\tau]\!]\eta[[\![\sigma]\!]\eta/\alpha] = [\![(\forall\beta.\tau)[\sigma/\alpha]]\!]\eta \quad \textit{and} \quad [\![\exists\beta.\tau]\!]\eta[[\![\sigma]\!]\eta/\alpha] = [\![(\exists\beta.\tau)[\sigma/\alpha]]\!]\eta.$$

3.2 Implicit polymorphism

We now present an interpretation of dynamic and recursive types in a language with implicit polymorphism in the style of *ML*. This gives a denotational interpretation of A. Mycroft's[12] proposal to extend the functional language *ML*. He pointed out that functions one would like to have for *ML*, like

$$print\ : \texttt{dyn} \to \texttt{string} \qquad \text{or} \qquad eval\ : \texttt{expression} \times \texttt{environment} \to \texttt{dyn},$$

could be defined when *ML* had a type `dyn`.

The previous notion of *types* is modified by adding universal type quantifiers in prenex form only, according to the grammar

$$
\begin{aligned}
\tau &= \alpha \mid \texttt{bool} \mid \texttt{nat} \mid \texttt{dyn} \\
&\quad \mid (\tau + \tau) \mid (\tau \times \tau) \mid (\tau \to \tau) \mid \mu\alpha.\tau && \textit{(monotypes)} \\
\overline{\sigma} &= \tau \mid \forall\alpha.\overline{\sigma}, && \textit{(polytypes)}.
\end{aligned}
$$

Let *MType* and *PType* be the set of *well-formed* mono- and polytypes, respectively. By *clMType* and *clPType* we mean the *closed* well-formed mono- and polytypes, respectively.

Terms are modified in that (**dynamic** $t : \tau$) is replaced by (**dynamic** t), and $\lambda x : \tau.t$ by $\lambda x.t$, and so terms do not contain type information any more, except in type patterns.

Semantics of types According to Milner's[11] interpretation for implicit polymorphism, type quantifiers are meant to range over closed monotypes only. In the ideal model, separate a universe $\mathcal{MT}$ of monotypes from the universe $\mathcal{T}$ of all types by

$$\mathcal{MT} := \{[\![\tau]\!] \mid \tau \in \mathit{clMType}\} \;\subseteq\; \mathcal{T} = \{J \in \mathcal{I} \mid \mathit{error}^V \notin J\}.$$

The meaning of polytypes is reduced to that of monotypes by induction on the quantifier-rank, using

$$[\![\forall\alpha.\,\overline{\sigma}]\!]\eta := \bigcap\{[\![\overline{\sigma}[\tau/\alpha]]\!]\eta \mid \tau \in \mathit{clMType}\}. \tag{3}$$

In order to cover type-tags with quantifiers, Definition 12 is changed by

$$[\![\texttt{dyn}]\!]\eta = \bigcup\{[\![\overline{\sigma}]\!]\eta * \{\overline{\sigma}\} \mid \overline{\sigma} \in \mathit{clPType}\}. \tag{4}$$

Theorem 22. *There is a meaning function $[\![\tau]\!]\eta$ for implicitly polymorphic types satisfying the conditions of Definition 12 and equation (3).*

Proof. Again, we modify the function F from the proof of Theorem 13 by adding component functions $F_{\forall\alpha.\overline{\sigma}}$ for polytypes $\overline{\sigma}$. The former *clType* has to be replaced by *clPType* everywhere. We define

$$F_{\forall\alpha.\overline{\sigma}}(I,\eta) := \bigcap\{F_{\overline{\sigma}[\tau/\alpha]}(I,\eta) \mid \tau \in \mathit{clMType}\}. \tag{5}$$

Claim 7. $F_{\forall\alpha.\overline{\sigma}}$ is 1/2-contractive in its first argument.

The proof of this is analogous to that of Claim 6, using Proposition 10. It follows that $\lambda I \in \Pi\mathcal{I}.F(I,\eta)$ is contractive, whence the meaning of types can again be defined by $[\![\overline{\sigma}]\!]\eta := I_\eta(\overline{\sigma})$, using the unique fixed point $I_\eta = F(I_\eta,\eta)$ of F.

The substitution lemma holds in the following form:

Proposition 23. *For each monotype τ, $[\![\overline{\sigma}]\!]\eta[[\![\tau]\!]\eta/\alpha] = [\![\overline{\sigma}[\tau/\alpha]]\!]\eta$.*

Proof. By induction on the quantifier-rank of $\overline{\sigma}$. Let θ be $\eta[[\![\tau]\!]\eta/\alpha]$. In the case of $\forall\beta.\overline{\sigma}$, we may assume $\alpha \not\equiv \beta$ and $free(\tau) \cap bound(\forall\beta.\overline{\sigma}) = \emptyset$, and thus

$$\begin{aligned}
[\![\forall\beta.\overline{\sigma}]\!]\eta[[\![\tau]\!]\eta/\alpha] &= F_{\forall\beta.\overline{\sigma}}(I_\theta, \theta) \\
&= \bigcap\{\, F_{\overline{\sigma}[\rho/\beta]}(I_\theta,\theta) \mid \rho \in clMType \,\} && \text{(by definition)} \\
&= \bigcap\{\, [\![\overline{\sigma}[\rho/\beta]]\!]\theta \mid \rho \in clMType \,\} && \text{(by definition)} \\
&= \bigcap\{\, [\![\overline{\sigma}[\rho/\beta][\tau/\alpha]]\!]\eta \mid \rho \in clMType \,\} && \text{(by induction)} \\
&= \bigcap\{\, [\![\overline{\sigma}[\tau/\alpha][\rho/\beta]]\!]\eta \mid \rho \in clMType \,\} && \text{(since } \rho \text{ is closed,} \\
&= [\![\forall\beta(\overline{\sigma}[\tau/\alpha])]\!]\eta && \beta \notin free(\tau)) \\
&= [\![(\forall\beta.\overline{\sigma})[\tau/\alpha]]\!]\eta. && \text{(by variable disjointness)}
\end{aligned}$$

Corollary 24. $[\![\forall\alpha.\overline{\sigma}]\!]\eta = \bigcap_{J\in\mathcal{MT}} [\![\overline{\sigma}]\!]\eta[J/\alpha]$.

Remark. In the absence of recursive types, one can avoid to define the meaning function for types by approximations, by using the (finite) recursive definition

$$\begin{aligned}
[\![\mathtt{dyn}]\!]\eta = {} & [\![\mathtt{bool}]\!]\eta * \{\mathtt{bool}\} \ \cup\ [\![\mathtt{nat}]\!]\eta * \{\mathtt{nat}\} \ \cup\ [\![\mathtt{dyn}]\!]\eta * \{\mathtt{dyn}\} \\
& \cup ([\![\mathtt{dyn}]\!]\eta \mathbin{\dot{+}} [\![\mathtt{dyn}]\!]\eta) \ \cup\ ([\![\mathtt{dyn}]\!]\eta \mathbin{\dot{\times}} [\![\mathtt{dyn}]\!]\eta) \ \cup\ ([\![\mathtt{dyn}]\!]\eta \mathbin{\dot{\rightarrow}} [\![\mathtt{dyn}]\!]\eta) \\
& \cup \dot{\forall}([\![\mathtt{dyn}]\!]\eta)
\end{aligned}$$

For an ideal I over $\mathcal{V}$, we define $\dot{\forall}(I)$ to be

$$\{\langle i, \forall\alpha.\overline{\sigma}\rangle^V \mid \forall\alpha.\overline{\sigma} \in clPType,\ \langle i, \overline{\sigma}[\tau/\alpha]\rangle^V \in I \text{ for all } \tau \in clMType\} \cup \{\bot_V\}.$$

In contrast to $\dot{+}$, $\dot{\times}$, and $\dot{\rightarrow}$, the operation $\dot{\forall}$ is *not* a contractive mapping on the space $(\mathcal{I}, d)$ of ideals of $\mathcal{V}$ – it is just non-expanding. To ensure that the recursion equation for $[\![\mathtt{dyn}]\!]$ has a solution, we modify the metric d on $\mathcal{V}$ as follows:

For polytypes $\overline{\sigma}$, let $qrk(\overline{\sigma})$, the *quantifier-rank of* $\overline{\sigma}$, be the number of (leading) quantifiers in $\overline{\sigma}$. For $v \in V$ and $I, J \in \mathcal{I}$, define a modified rank and distance by

$$\begin{aligned}
\widetilde{rk}(v) &:= \begin{cases} rk(v) + qrk(\overline{\sigma}) & \text{if } v\lceil_{(V*clType)} = \langle i, \overline{\sigma}\rangle \\ rk(v), & \text{otherwise,} \end{cases} \\
\widetilde{d}(I,J) &:= 2^{-min\{\widetilde{rk}(v) \mid v \in I \bowtie J\}}.
\end{aligned}$$

The reader may check that the functions $+$, $\times$, $\rightarrow$, $\dot{+}$, $\dot{\times}$, $\dot{\rightarrow}$, $*\{\tau\}$, and $\dot{\forall}$ are contractive on $(\mathcal{I}, \widetilde{d})$.

Semantics of terms One might wish to define the meaning of (**dynamic** e) in terms of the principal type of e, but e need not have a principal type: for example, $e = \lambda x.$(**dynamic** x) is of type $\mathtt{dyn} \rightarrow \mathtt{dyn}$ and $\mathtt{nat} \rightarrow \mathtt{dyn}$, but has no principal type. For type inference, use the rules of *ML* together with the implicit version (Dyn E) of `dyn`-elimination and the following implicit version of `dyn`-introduction:

$$\text{(Dyn I)} \quad \frac{\Gamma \ \triangleright\ e : \tau}{\Gamma \ \triangleright\ (\textbf{dynamic } e) : \mathtt{dyn}}, \quad \text{if } \overline{\tau}^{\Gamma} \text{ is closed.}$$

Since terms t lack principal types, the meaning $[\![t]\!]\eta$ can only be given relative to a typing derivation D for t. For each subterm (**dynamic** e) of t, D uniquely fixes a closed type $\overline{\tau}^{\Gamma}$ that can be used for tagging the value of e. To do so, we assign to each subterm r of t a sequence π of numbers $i \in \{1, 2, 3\}$, coding the branch leading from the root to r in the tree representation of t. Then we can define

$$[\![(\mathbf{dynamic}\ e^{\pi 1})^{\pi}]\!]\eta := \begin{cases} error^V, & \text{if } [\![e^{\pi 1}]\!]\eta = error^V \\ \langle [\![e^{\pi 1}]\!]\eta, \overline{\tau}^{\Gamma} \rangle^V, & \text{if } \Gamma \rhd e^{\pi 1} : \tau \text{ is the inference step of } D \\ & \text{that assigns a type to } e^{\pi 1}. \end{cases}$$

To define the meaning of typecase-terms, we match polytypes as follows. For $\overline{\sigma} \equiv \forall \gamma_1 \ldots \gamma_n . \sigma(\gamma_1, \ldots, \gamma_n) \in clType$ and $\overline{\tau}(\alpha) \equiv \forall \beta_1 \ldots \beta_m . \tau(\alpha, \beta_1, \ldots, \beta_m) \in PType$, define

$$match_{\{\alpha\}}(\overline{\sigma}, \overline{\tau}) = [\rho/\alpha] :\Longleftrightarrow \rho \text{ is closed and } \sigma[\rho_1/\gamma_1, \ldots, \rho_n/\gamma_n] \equiv \tau[\rho/\alpha],$$

where $[\rho/\alpha, \rho_1/\gamma_1, \ldots, \rho_n/\gamma_n]$ is the most general unifier of monotypes σ and τ when considering the β_i in τ as constants. Since every instantiation of $\tau[\rho/\alpha]$ by closed monotypes is an instance of σ by closed monotypes as well, $[\![\overline{\sigma}]\!] \subseteq [\![\forall \beta_1 \ldots \beta_m . \tau[\rho/\alpha]]\!]$.

Leaving out the path annotation which relativize the meaning to D, we can now define the meaning of typecase-expressions just as in Section 2.4:

$$[\![(\mathbf{typecase}\ d\ \mathbf{of}\ \{\alpha\}\ x : \overline{\tau}\ \mathbf{then}\ r\ \mathbf{else}\ s)]\!]\eta :=$$
$$\begin{cases} [\![r[\rho/\alpha]]\!]\eta[v/x], & \text{if } \langle v, \overline{\sigma} \rangle = [\![d]\!]\eta\lceil_{(V * clType)} \text{ and } match_{\{\alpha\}}(\overline{\sigma}, \overline{\tau}) = [\rho/\alpha] \\ [\![s]\!]\eta, & \text{if } \langle v, \overline{\sigma} \rangle = [\![d]\!]\eta\lceil_{(V * clType)} \text{ and } match_{\{\alpha\}}(\overline{\sigma}, \overline{\tau}) = fail, \\ & \quad \text{or } [\![d]\!]\eta = \bot_V \\ error^V, & \text{else.} \end{cases}$$

Observe that pattern-variables α in patterns range over *closed* types ρ only; if ρ were allowed to contain free variables (bound by the quantifiers of the pattern or the type-tag), it would be unclear what α means in the then-branch of the typecase-statement. Compared with the treatment of non-closed patterns in Section 3 of [8], the above corresponds to their case of patterns with existential type quantifier prefixes.

In the presence of recursive types, it seems preferable that in the definition of the *match*-function obove we read $\equiv$ not as syntactical identity of type expressions, but rather as identity of the rational trees obtained by infinite unfolding of the μ-operator (c.f. [4]). The fold and unfold-rules for recursive types should then be replaced by the stronger rule of equality for recursive types, see [4].

4 Open problems

By providing an interpretation in the ideal model, it has been shown that dynamic types can be combined with recursive types and explicit or implicit polymorphism in a sound way. This gives a partial answer to questions of Abadi e.a.[1], and adds semantical support to implementations integrating dynamic types into polymorphic languages like *CAML*[5].

The main drawbacks of the extension of *ML* by dynamic typing sketched in Section 3.2 and the similar ones of [2] and [8] are the failure of the principal types property and the restriction to closed type-tags. For languages where types can be passed as parameters, models for dynamics with *open* type-tags are needed.

An open point is to remove the well-formedness restriction on types. Domains with a notion of approximation were introduced by Cardone and Coppo[4] in order to give meanings to arbitrary –not just contractive– recursive types. We believe that `dyn` can be added to the simple and recursive types of [4]. But we do not know whether the completeness result of [4] carries over to the system of Section 2.1.

For typecase-expressions with higher-order matching as in [2], or just the systems of Section 3 above, even soundness theorems have not yet been given.

Acknowledgement I wish to thank the referees for some very helpful proposals to improve the results and presentation. Thanks also to Fritz Henglein for sending a copy of Mycroft's papers and for a hint to [4].

References

1. M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically-typed language. In *16th POPL*, pages 213–227, 1989.
2. M. Abadi, L. Cardelli, B. Pierce, and D. Remy. Dynamic typing in polymorphic languages. In *ACM SIGPLAN Workshop on ML and its Applications. San Francisco, California, June 20-21, 1992*, pages 92–103, 1992.
3. L. Cardelli. Amber. In G. Cousineau, P. L. Curien, and B. Robinet, editors, *Combinators and Functional Programming Languages.* Springer LNCS 242, 1986.
4. F. Cardone and M. Coppo. Type inference with recursive types: Syntax and semantics. *Information and Computation*, 92(1):48–80, May 1991.
5. G. Cousineau and G. Huet. The CAML Primer. Version 2.6. Project Formel, INRIA-ENS, April 1989.
6. R. Harper, R. Milner, and M. Tofte. The definition on Standard ML - Version 2. LFCS Report Series ECS-LFCS-88-62, Dept. of Computer Science, Univ. of Edinburgh, 1988.
7. F. Henglein. Dynamic typing. In *European Symposium on Programming (ESOP). Rennes, France*, pages 233–253. Springer LNCS, vol. 582, 1992.
8. X. Leroy and M. Mauny. Dynamics in ML. In *Conf. on Functional Programming Languages and Computer Architecture. Cambridge, Massachusetts, August 1991*, pages 406–426. Springer LNCS 523.
9. D. MacQueen, G. Plotkin, and R. Sethi. An ideal model for recursive polymorphic types. In *Proceedings of the 11th ACM Symposium on Principles of Programming Languages*, 1984.
10. D. C. J. Matthews. Static and Dynamic Type-Checking. In: Papers on Poly/ML. Technical Report 161, Computer Laboratory, University of Cambridge, February 1989.
11. R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.
12. A. Mycroft. Dynamic types in statically typed languages (preliminary draft). Unpublished typescript, December 1983.
13. A. Mycroft. Dynamic types in statically typed languages (2nd draft version). Unpublished typescript, August 1984.