FAITHFUL IDEAL MODELS FOR RECURSIVE POLYMORPHIC TYPES*

MARTÍN ABADI
Digital Equipment Corporation
Systems Research Center
130 Lytton Avenue
Palo Alto, CA 94301, USA

BENJAMIN PIERCE†
School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213, USA

GORDON PLOTKIN‡
Department of Computer Science
King's Building
University of Edinburgh
Edinburgh, EH9 3JZ, UK

ABSTRACT

We explore ideal models for a programming language with recursive polymorphic types, variants of the model studied by MacQueen, Plotkin, and Sethi. The use of suitable ideals yields a close fit between models and programming language. Two of our semantics of type expressions are faithful, in the sense that programs that behave identically in all contexts have exactly the same types.

Keywords: Polymorphism, recursive types, full abstraction, ideals, metric models.

1. Introduction

Often, a formal semantics assigns different values to programs that behave identically in all contexts [1, 2, 3]. In other words, the semantics of programming-language expressions is not fully abstract. This mismatch between model and programming language elicits diverse reactions. Some propose extensions to the programming language, while others prefer modifying the semantics. Both of these attitudes have suggested fruitful lines of research. Yet others point out that the semantics still serves its purposes fairly well, for example in proving the safety of evaluation schemes. At any rate, full abstraction is one of the main criteria in assessing programming-language semantics [4].

A similar situation commonly arises in the semantics of type systems. In its simplest form, a semantics for a type system maps each type expression to a reasonable subset of the domain of values. Let us call these subsets types. In most

*A preliminary version of this paper was presented at the 4th IEEE Symposium on Logic in Computer Science in June 1989.

†This work was started at Digital Equipment Corporation, Systems Research Center.

‡This work was started at the Center for the Study of Language and Information, Stanford University.

semantics, some programs that behave identically in all contexts not only may receive different meanings—they may even belong to different types. Roughly, we call a semantics *faithful* if programs that behave identically in all contexts have identical types.

In logical form, the faithfulness problem concerns the soundness of the usual rule of inference

$$\frac{e : T \qquad e' = e}{e' : T}$$

which says that if e has type T and e' and e are "equal" then e' has type T. In the standard reading of this rule, two expressions are "equal" when they are equal in a given model. We take two expressions to be "equal" when they behave identically in all contexts. A faithful semantics should validate the rule under this interpretation.

Clearly, a semantics of type expressions is automatically faithful if the underlying semantics of expressions is fully abstract. The converse is false, however, and it seems intriguing (and perhaps easier) to study faithfulness in isolation from full abstraction.

In this paper, we give semantics for a language with recursive polymorphic types, like Standard ML [5]. Imitating MacQueen, Plotkin, and Sethi, we adopt and extend the ideal model [6]. In the ideal model, type expressions are interpreted as ideals, certain subsets of the universe of values. We place new restrictions on ideals, defining the classes of generated ideals, coarse ideals, and abstract ideals. In our variants of the model, type expressions are interpreted as ideals in these classes. We obtain faithful semantics of type expressions. These semantics validate the stronger rule of inference

$$\frac{e : T \qquad e' \sqsubseteq e}{e' : T}$$

which says that if e has type T and e' is "less than" e then e' has type T, with "less than" interpreted contextually (just as "equal" above).

One danger in tampering with a model is making it mathematically intractable, for instance by making it too syntactic too soon. We may also simply give up useful properties. Fortunately, our type systems share many features with the one explored by MacQueen, Plotkin, and Sethi. In particular, a large family of recursive type equations have unique solutions.

The departures from the original ideal model are perhaps best illustrated with an example. Consider the following expression of the untyped lambda calculus:

```
explode-||-or = λ x.
                   if  x(true, btm)
                   and x(btm, true)
                   and not x(false, false)
                   then wrong
                   else true
```

where `btm` is a divergent expression, such as `(λ y. y y) (λ y. y y)`, and `wrong` represents a dynamic type error.

How should we type the expression `explode-||-or`? The first branch of the `if-then-else` is only taken when the input represents the parallel-or function, which returns `true` whenever either of its arguments is `true`, even if the other argument diverges. In many common models, and in the one considered by MacQueen, Plotkin, and Sethi in particular, the parallel-or function exists. In these models, `explode-||-or` returns `wrong` with a binary boolean function as input, and hence it seems ill-typed. On the other hand, it is well-known that the parallel-or function cannot be expressed in the usual lambda calculus, as studied by Plotkin [2]. Therefore, whenever `explode-||-or` is applied to an expression in the language that represents a binary boolean function, `explode-||-or` returns `true` if it terminates. Thus, intuitively, there seem to be grounds for asserting that `explode-||-or` should have type $(bool \times bool \rightarrow bool) \rightarrow bool$, the type of functions that map binary boolean functions to boolean values. This is the case in our semantics.

In the next section we describe the setting for this work: a language, a model, the semantics of the language in the model, and some useful expressions of the language. We also define faithfulness precisely. In Section 3 we study the generated ideals; this section contains analogues to the central results in [6]. In Section 4 we use the generated ideals to give a semantics to type expressions, and then restrict our attention to special classes of generated ideals; we prove that two semantics are faithful. Finally, in Section 5, we consider more syntactic approaches, where types are sets of terms.

2. The Setting

We work within the ideal model of types, following MacQueen, Plotkin, and Sethi. The setting is almost exactly identical to theirs. We refer the reader to their paper for definitions and lemmas that we omit or only sketch, and recommend the study of their paper as a preliminary for fully understanding the constructions below.

In this section we briefly review a simple programming language, a model, and the denotational semantics of the programming language in the model. Then we define faithfulness. We also study some expressions of the programming language that play a role in the formulation of syntactic counterparts to semantic concepts.

2.1. The Programming Language

The basic programming language is a simple untyped lambda calculus with constants. We start with the following grammar:

$$\mathtt{e} ::= \mathtt{c} \mid \mathtt{x} \mid \lambda\mathtt{x}.\mathtt{e} \mid \mathtt{e(e')}$$

Here, `e` and `e'` range over the set $\textsc{Exp}$ of expressions, `c` over a suitable set of constants, and `x` over the set $\textsc{Var}$ of variables. We insist on having a sufficiently rich set of constants, such as the one considered in [6]: `true`, `false`, `cond`, `0`, `z` (test for zero), `+1`, `-1`, `pair`, π_1 and π_2 (for extracting pair elements), `inl` and `inr` (for forming sums), `outl` and `outr` (to recover elements from sums), and `isl` and `isr` (for distinguishing cases in sums). We also use abbreviations, such as `and`, `if-then-else`, and `x < m` (with `m` fixed).

The only new feature of our language is an additional `cases` construct. Intuitively, this construct enables us to decide whether $\mathtt{e}_{\mathtt{sel}}$ is a boolean, a natural, a pair, a sum, or a function, and to return a different result in each of the cases:

```
cases e_sel
 bool: e_bool
 nat:  e_nat
 pair: e_pair
 sum:  e_sum
 fun:  e_fun
end
```

It can be argued that the addition of `cases` is natural, and even that `cases` either should be definable or ought to have been in the programming language in the first place. At any rate, we use `cases` only for writing a limited family of expressions. It is common to assume that similar expressions are definable [1], and we include `cases` only to make their definition straightforward.

2.2. The Model

To give a semantics to this language, we consider a universe $\mathbf{V}$ that satisfies the isomorphism equation

$$\mathbf{V} \cong \mathbf{T} + \mathbf{N} + (\mathbf{V}\times\mathbf{V}) + (\mathbf{V}+\mathbf{V}) + (\mathbf{V}\rightarrow\mathbf{V}) + \mathbf{W},$$

where $\mathbf{T}$ is the flat domain of boolean values, $\mathbf{N}$ is the flat domain of natural numbers, $\mathbf{W}$ is the type-error domain $\{w\}_\perp$, and $\times$, $+$, and $\rightarrow$ represent the usual product, coalesced sum, and continuous function space operations, respectively (coalesced sum identifies the least elements of its two arguments). The value w represents all dynamic type errors.

The universe $\mathbf{V}$ can be constructed as the limit of a sequence of approximations $\mathbf{V}_0$, $\mathbf{V}_1$, ..., where

$$\mathbf{V}_0 = \{\perp\}$$

and, for all i,

$$\mathbf{V}_{i+1} = \mathbf{T} + \mathbf{N} + (\mathbf{V}_i\times\mathbf{V}_i) + (\mathbf{V}_i+\mathbf{V}_i) + (\mathbf{V}_i\rightarrow\mathbf{V}_i) + \mathbf{W}.$$

We omit the details of the construction, which are standard [7, 6].

As usual, we have an ordering $\sqsubseteq$ on $\mathbf{V}$; we read $x \sqsubseteq y$ as "x is less defined than y" or "x approximates y." An element of $\mathbf{V}$ is finite if whenever it approximates the least upper bound of a chain it approximates some element of the chain. The function μ_i is the embedding from $\mathbf{V}_i$ to $\mathbf{V}$, and μ_i^{R} is its inverse projection. The rank $r(a)$ of a finite element a of $\mathbf{V}$ is the least i such that a "appears" in $\mathbf{V}_i$ during the construction of $\mathbf{V}$ as a limit; the rank function is only defined on finite elements. More precisely, $r(a)$ equals the least i such that $a = \mu_i \circ \mu_i^{\mathrm{R}}(a)$, provided a is finite.

2.3. The Semantics of Expressions

We assign a meaning $[\![\mathtt{e}]\!]$ to each expression `e` of the programming language. The definition of $[\![\ ]\!]$ appears in Fig. 1, with the following notation:

- s in $\mathbf{V}$, where s belongs to a summand $\mathbf{S}$ of $\mathbf{V}$, is the injection of s into $\mathbf{V}$;

- $\mathtt{wrong}$ is an abbreviation for w in $\mathbf{V}$;
- if $v = (s \text{ in } \mathbf{V})$ for some $s \in \mathbf{S}$ then $v|_{\mathbf{S}}$ is s, and otherwise it is $\perp$;
- $v \sqsubseteq \mathbf{S}$ yields $\perp$ if v is $\perp$, true if $v = (s \text{ in } \mathbf{V})$ for some $s \in \mathbf{S}$ (with $s \neq \perp$), and false otherwise;
- a mapping in $\textsc{Var} \rightarrow \mathbf{V}$ is called an environment; $\rho\{\mathtt{x} \leftarrow v\}$ is the environment obtained from ρ by giving the value v to the variable $\mathtt{x}$.

We omit the equations for constants.

$$[\![\ ]\!] : \textsc{Exp} \rightarrow (\textsc{Var} \rightarrow \mathbf{V}) \rightarrow \mathbf{V}$$

$$\begin{array}{lcl}
[\![\mathtt{x}]\!]_\rho & = & \rho(\mathtt{x}) \\
[\![\lambda\mathtt{x.e}_{\mathtt{body}}]\!]_\rho & = & (\lambda v.[\![\mathtt{e}_{\mathtt{body}}]\!]_{\rho\{x \leftarrow v\}}) \text{ in } \mathbf{V} \\
[\![\mathtt{e}_{\mathtt{fun}}(\mathtt{e}_{\mathtt{arg}})]\!]_\rho & = & \text{let } v = [\![\mathtt{e}_{\mathtt{fun}}]\!]_\rho \text{ in} \\
 & & \quad \text{if } v \sqsubseteq (\mathbf{V} \rightarrow \mathbf{V}) \text{ then } ([\![\mathtt{e}_{\mathtt{fun}}]\!]_\rho|_{\mathbf{V} \rightarrow \mathbf{V}})([\![\mathtt{e}_{\mathtt{arg}}]\!]_\rho) \\
 & & \quad \text{else } \mathtt{wrong} \\
\multicolumn{3}{l}{[\![\mathtt{cases}\ \mathtt{e}_{\mathtt{sel}}\ \mathtt{bool:}\ \mathtt{e}_{\mathtt{bool}}\ \ldots]\!]_\rho} \\
 & = & \text{let } v = [\![\mathtt{e}_{\mathtt{sel}}]\!]_\rho \text{ in} \\
 & & \quad \text{if } v \sqsubseteq \mathbf{T} \text{ then } [\![\mathtt{e}_{\mathtt{bool}}]\!]_\rho \\
 & & \quad \text{else if } \ldots \\
 & & \quad \text{else } \mathtt{wrong}
\end{array}$$

Fig. 1: Definition of the meaning function for expressions

In our constructions, we are particularly interested in denotable elements, that is, the elements in the range of $[\![\ ]\!]$:

Definition 2.3.1. *An element $v \in \mathbf{V}$ is denotable if $v = [\![\mathtt{e}]\!]$ for some closed expression* $\mathtt{e}$.

In the rest of this paper, as in this definition, we often do not mention the environment ρ when it is irrelevant. Specifically, if $\mathtt{e}$ is a closed expression then, informally, we may write $[\![\mathtt{e}]\!]$ instead of $[\![\mathtt{e}]\!]_\rho$, since $[\![\mathtt{e}]\!]_\rho$ clearly does not depend on ρ.

2.4. Faithfulness

At this point, a precise discussion of full abstraction and faithfulness is in order.

Roughly, a semantics of expressions is fully abstract if two expressions that behave identically in all contexts receive the same interpretation. We could be interested in different aspects of the behavior of an expression. We find it sufficient here to "observe termination," that is, to identify the behavior of $\mathtt{e}$ with the set of all contexts $\mathcal{C}$ such that $[\![\mathcal{C}\mathtt{[e]}]\!] \neq \perp$—or, equivalently, with the set of all contexts $\mathcal{C}$ such that $[\![\mathcal{C}\mathtt{[e]}]\!] = \perp$. In other words, we say that two expressions $\mathtt{e}$ and $\mathtt{e'}$ behave identically if, for all contexts $\mathcal{C}$, we have $[\![\mathcal{C}\mathtt{[e]}]\!] = \perp$ if and only if $[\![\mathcal{C}\mathtt{[e']}]\!] = \perp$. For the definition to make sense, we consider only the contexts such that $\mathcal{C}[\mathtt{e}]$ and $\mathcal{C}[\mathtt{e'}]$ are closed; we leave similar requirements implicit in the rest of the paper.

Other notions of observation are also justifiable; for instance, we might care about the set of contexts $\mathcal{C}$ such that $[\![\mathcal{C}[\mathtt{e}]]\!]$ is a boolean value. Fortunately, many reasonable kinds of observation are equivalent to the one we choose (see [4] for an informal discussion of this point).

Thus, full abstraction requires that, for all $\mathtt{e}$ and $\mathtt{e'}$, whenever $[\![\mathcal{C}[\mathtt{e}]]\!] = \perp$ if and only if $[\![\mathcal{C}[\mathtt{e'}]]\!] = \perp$ for all contexts $\mathcal{C}$, it must also be the case that $[\![\mathtt{e}]\!] = [\![\mathtt{e'}]\!]$. An interesting refinement of this definition allows the ordering relation $\sqsubseteq$ to come into play. In this refinement, for all $\mathtt{e}$ and $\mathtt{e'}$, whenever $[\![\mathcal{C}[\mathtt{e}]]\!] = \perp$ implies $[\![\mathcal{C}[\mathtt{e'}]]\!] = \perp$ for all contexts $\mathcal{C}$, it must also be the case that $[\![\mathtt{e'}]\!] \sqsubseteq [\![\mathtt{e}]\!]$. (The converse holds trivially.)

Let us write $\mathtt{e} \models \mathcal{C}$ if $[\![\mathcal{C}[\mathtt{e}]]\!] \neq \perp$, and $\mathtt{e'} \underset{\sim}{\sqsubseteq} \mathtt{e}$ if $\mathtt{e'} \models \mathcal{C}$ implies $\mathtt{e} \models \mathcal{C}$ for all $\mathcal{C}$. In this notation, the refined definition of full abstraction simply says that $\underset{\sim}{\sqsubseteq}$ and $\sqsubseteq$ are identical on denotable elements.

Note that, in either case, as in [1], full abstraction is a purely semantic criterion and does not refer to a particular operational semantics. This is a convenient way to consider the full-abstraction property as an intrinsic feature of a semantics.

Analogously, a semantics of type expressions is faithful if two expressions that behave identically in all contexts have identical types. For this, it suffices that each type be abstract, in the sense that membership in the type does not distinguish between expressions that behave identically in all contexts. This tentative definition can also be refined to take the ordering relation into account, and motivates a stronger definition:

Definition 2.4.1. *A set I is abstract if, for all closed expressions $\mathtt{e}$ and $\mathtt{e'}$, if $\mathtt{e'} \underset{\sim}{\sqsubseteq} \mathtt{e}$ and $[\![\mathtt{e}]\!] \in I$ then $[\![\mathtt{e'}]\!] \in I$.*

In an expanded, symbolic form, a set I is abstract if

$$\forall \mathtt{e}, \mathtt{e'}.[\forall \mathcal{C}.([\![\mathcal{C}[\mathtt{e'}]]\!] = \perp \supset [\![\mathcal{C}[\mathtt{e}]]\!] = \perp) \supset ([\![\mathtt{e'}]\!] \in I \supset [\![\mathtt{e}]\!] \in I)].$$

Informally, the abstract sets are "closed downwards" in $\underset{\sim}{\sqsubseteq}$. This requirement follows from common intuitions on the structure of types, the same intuitions that underlie the use of ideals (we come back to this point in Section 3).

Finally, we can define faithfulness. The definition implies the one we have been using informally.

Definition 2.4.2. *Given a set of well-formed type expressions, a semantics is faithful if the meaning of each well-formed type expression is abstract.*

In logical terms, then, we require the soundness of the rule

$$\frac{e : T \qquad e' \sqsubseteq e}{e' : T}$$

with $\sqsubseteq$ interpreted as $\underset{\sim}{\sqsubseteq}$.

The following expressions are useful in establishing correspondences between syntax and semantics. In particular, they are related to the rank function. We use these expressions and some of their properties (given in Lemma 2.5.1) in the main proofs below.

First we define a sequence of expressions $\mathtt{p}^{\mathtt{m}}_{\mathtt{n}}$:

```
p^m_0 = btm

p^m_{n+1} = λ x.
        cases x
         bool: x
         nat:  if x < m then x else btm
         pair: pair(p^m_n(π_1 x), p^m_n(π_2 x))
         sum:  if isl(x) then inl(p^m_n(outl x))
                          else inr(p^m_n(outr x))
         fun:  p^m_n ∘ x ∘ p^m_n
        end
```

where, as before, `btm` is a divergent expression, that is, an expression that represents $\perp$, such as `(λ y. y y) (λ y. y y)`. Intuitively, each $\mathtt{p}^{\mathtt{m}}_{\mathtt{n}}$ is a projection that maps an object to an approximation with smaller rank.

We also define a sequence of expressions $\mathtt{p}_{\mathtt{n}}$:

```
p_0 = btm

p_{n+1} = λ x.
        cases x
         bool: x
         nat:  x
         pair: pair(p_n(π_1 x), p_n(π_2 x))
         sum:  if isl(x) then inl(p_n(outl x))
                          else inr(p_n(outr x))
         fun:  p_n ∘ x ∘ p_n
        end
```

Intuitively, each $\mathtt{p}_{\mathtt{n}}$ is the limit of the expressions $\mathtt{p}^{\mathtt{m}}_{\mathtt{n}}$ as `m` grows, and is also a projection. The limit of the expressions $\mathtt{p}_{\mathtt{n}}$ as `n` grows is the identity function.

Lemma 2.5.1. *Let id be the identity function in* $\mathbf{V}$*; for all* `m` *and* `n`*,*

1. $[\![\mathtt{p}^{\mathtt{m}}_{\mathtt{n}}]\!]$ *has a finite range;*
2. $[\![\mathtt{p}^{\mathtt{m}}_{\mathtt{n}}]\!] \sqsubseteq id$*;*
3. $[\![\mathtt{p}^{\mathtt{m}}_{\mathtt{n}}]\!] \circ [\![\mathtt{p}^{\mathtt{m}}_{\mathtt{n}}]\!] = [\![\mathtt{p}^{\mathtt{m}}_{\mathtt{n}}]\!]$*;*

4. *all elements in the range of* $[\![p^{m}_{n}]\!]$ *are finite;*

5. $[\![p^{m}_{n}]\!] \sqsubseteq [\![p^{m+1}_{n}]\!]$ *and* $[\![p^{m}_{n}]\!] \sqsubseteq [\![p^{m}_{n+1}]\!]$*;*

6. $[\![p_{n}]\!] = \bigsqcup_{m} [\![p^{m}_{n}]\!]$*;*

7. $[\![p_{n}]\!] = \mu_n \circ \mu^R_n$*;*

8. $id = \bigsqcup_{n} [\![p_{n}]\!] = \bigsqcup_{n} [\![p^{n}_{n}]\!]$*;*

9. $[\![p_{n}]\!] \circ [\![p^{m}_{n}]\!] = [\![p^{m}_{n}]\!]$*.*

Proof We prove the claims in order.

1. $[\![\mathtt{p}^{m}_{n}]\!]$ has a finite range: The proof is by induction on `n`. The case `n` $=$ `0` is trivial. Assume that the proposition holds for some `n`, to check it for `n+1`. The range of $[\![\mathtt{p}^{m}_{n+1}]\!]$ consists of numbers less than `m`, pairs and sums of elements in the range of $[\![\mathtt{p}^{m}_{n}]\!]$, and functions into the range of $[\![\mathtt{p}^{m}_{n}]\!]$ and determined by their values on the range of $[\![\mathtt{p}^{m}_{n}]\!]$. By the induction hypothesis, this set must be finite.

2. $[\![\mathtt{p}^{m}_{n}]\!] \sqsubseteq \mathrm{id}$: Again, we use induction on `n`. The base case is trivial. The inductive step follows immediately from the monotonicity of all functions.

3. $[\![\mathtt{p}^{m}_{n}]\!] \circ [\![\mathtt{p}^{m}_{n}]\!] = [\![\mathtt{p}^{m}_{n}]\!]$: Once more, we use induction on `n`. The base case is trivial. The inductive step follows from the remarks in 1 on the nature of $[\![\mathtt{p}^{m}_{n+1}]\!]$'s range.

4. All elements in the range of $[\![\mathtt{p}^{m}_{n}]\!]$ are finite: Suppose that for some `e` and for some chain $\langle y_l \rangle$ we have $[\![\mathtt{p}^{m}_{n}(\mathtt{e})]\!] \sqsubseteq \bigsqcup_l y_l$. Then $[\![\mathtt{p}^{m}_{n}(\mathtt{e})]\!] \sqsubseteq \bigsqcup_l [\![\mathtt{p}^{m}_{n}]\!](y_l)$ (by the continuity of $[\![\mathtt{p}^{m}_{n}]\!]$ and 3). Hence $[\![\mathtt{p}^{m}_{n}(\mathtt{e})]\!] \sqsubseteq [\![\mathtt{p}^{m}_{n}]\!](y_k)$ for some k (by 1), and, finally, $[\![\mathtt{p}^{m}_{n}(\mathtt{e})]\!] \sqsubseteq y_k$ for some k (by 2).

5. $[\![\mathtt{p}^{m}_{n}]\!] \sqsubseteq [\![\mathtt{p}^{m+1}_{n}]\!]$ and $[\![\mathtt{p}^{m}_{n}]\!] \sqsubseteq [\![\mathtt{p}^{m}_{n+1}]\!]$: The two propositions are proved separately, by induction on `m` and `n`, respectively. Both arguments are trivial

6. $[\![\mathtt{p}_{n}]\!] = \bigsqcup_{m} [\![\mathtt{p}^{m}_{n}]\!]$: First, $\bigsqcup_{m} [\![\mathtt{p}^{m}_{n}]\!]$ is defined, by 5. Furthermore, it is simple to prove by induction on `n` that $[\![\mathtt{p}_{n}]\!] \sqsubseteq \bigsqcup_{m} [\![\mathtt{p}^{m}_{n}]\!]$ (using continuity) and $\bigsqcup_{m} [\![\mathtt{p}^{m}_{n}]\!] \sqsubseteq [\![\mathtt{p}_{n}]\!]$ (using monotonicity).

7. $[\![\mathtt{p}_{n}]\!] = \mu_{\mathrm{n}} \circ \mu^{\mathrm{R}}_{\mathrm{n}}$: The proof is by induction on `n` and requires elementary properties of the embeddings μ_{n}. Specifically, it requires that $\mu_0 \circ \mu^{\mathrm{R}}_0$ is totally undefined, that $\mu_{\mathrm{n+1}} \circ \mu^{\mathrm{R}}_{\mathrm{n+1}}$ is the identity function on boolean values and numbers, that if f is a function then $\mu_{\mathrm{n+1}} \circ \mu^{\mathrm{R}}_{\mathrm{n+1}}(f) = (\mu_{\mathrm{n}} \circ \mu^{\mathrm{R}}_{\mathrm{n}}) \circ f \circ (\mu_{\mathrm{n}} \circ \mu^{\mathrm{R}}_{\mathrm{n}})$, and similar properties for pairs and sums. The argument is straightforward.

8. $\mathrm{id} = \bigsqcup_{n} [\![\mathtt{p}_{n}]\!] = \bigsqcup_{n} [\![\mathtt{p}^{n}_{n}]\!]$: The former equality follows from 7 while the latter one follows from 6.

9. $[\![\mathtt{p}_{n}]\!] \circ [\![\mathtt{p}^{m}_{n}]\!] = [\![\mathtt{p}^{m}_{n}]\!]$: This follows directly from 3 and 6, which implies that $[\![\mathtt{p}^{m}_{n}]\!] \sqsubseteq [\![\mathtt{p}_{n}]\!] \sqsubseteq \mathrm{id}$. □

These properties can now be used to prove the following lemma, which explains how to reduce the rank of a denotable value $[\![\mathtt{e}]\!]$ while remaining "above" a given lower bound a.

Lemma 2.5.2. *If $r(a) = \mathtt{n}$ and $a \sqsubseteq [\![\mathtt{e}]\!]$ then there exists $\mathtt{m}$ such that $a \sqsubseteq [\![\mathtt{p}^{\mathtt{m}}_{\mathtt{n}}(\mathtt{e})]\!] \sqsubseteq [\![\mathtt{e}]\!]$ and $r([\![\mathtt{p}^{\mathtt{m}}_{\mathtt{n}}(\mathtt{e})]\!]) \leq \mathtt{n}$.*

Proof This follows easily from Lemma 2.5.1.

Because a has rank $\mathtt{n}$, $[\![\mathtt{p}_{\mathtt{n}}]\!](a) = a$, and hence the monotonicity of $[\![\mathtt{p}_{\mathtt{n}}]\!]$ guarantees that $a \sqsubseteq [\![\mathtt{p}_{\mathtt{n}}]\!][\![\mathtt{e}]\!]$. In addition, since $[\![\mathtt{p}_{\mathtt{n}}]\!][\![\mathtt{e}]\!] = \bigsqcup_{\mathtt{k}}[\![\mathtt{p}^{\mathtt{k}}_{\mathtt{n}}]\!][\![\mathtt{e}]\!]$ and a is finite, there exists $\mathtt{m}$ such that $a \sqsubseteq [\![\mathtt{p}^{\mathtt{m}}_{\mathtt{n}}]\!][\![\mathtt{e}]\!]$, that is, $a \sqsubseteq [\![\mathtt{p}^{\mathtt{m}}_{\mathtt{n}}(\mathtt{e})]\!]$. Furthermore, $[\![\mathtt{p}^{\mathtt{m}}_{\mathtt{n}}(\mathtt{e})]\!] \sqsubseteq [\![\mathtt{e}]\!]$, since $[\![\mathtt{p}^{\mathtt{m}}_{\mathtt{n}}]\!]$ is a projection.

Finally, $[\![\mathtt{p}^{\mathtt{m}}_{\mathtt{n}}(\mathtt{e})]\!]$ is finite; Lemma 2.5.1 yields that $[\![\mathtt{p}_{\mathtt{n}}]\!][\![\mathtt{p}^{\mathtt{m}}_{\mathtt{n}}(\mathtt{e})]\!] = [\![\mathtt{p}^{\mathtt{m}}_{\mathtt{n}}(\mathtt{e})]\!]$, and hence $r([\![\mathtt{p}^{\mathtt{m}}_{\mathtt{n}}(\mathtt{e})]\!]) \leq \mathtt{n}$. □

3. On Tying Ideals to Language

In the original ideal model, two different ideals may contain exactly the same denotable elements. Hence these ideals cannot really be distinguished from the point of view of the programming language; their distinction may be regarded as irrelevant, or even bothersome. Analogous situations arise in other typical semantics.

In this section we study a notion of type with closer ties to the programming language. Each type can be obtained from the set of its denotable elements. Since these types are ideals, we call them denotably-generated ideals, or generated ideals for short. We define functions on generated ideals and prove that these functions share many of the desirable mathematical properties of the corresponding functions on arbitrary ideals.

3.1. Generated Ideals

As is commonly argued, types consist of sets of values with a common structure. Intuitively, the structural criteria embodied by type distinctions should be preserved "downwards" and "under limits" in the $\sqsubseteq$ ordering. This principle underlies the identification of types with ideals:

Definition 3.1.1. *A set I is an ideal if*

- *$I \neq \emptyset$;*
- *for all x and y, if $x \sqsubseteq y$ and $y \in I$ then $x \in I$; and*
- *for all increasing sequences $\langle x_n \rangle$, if $x_n \in I$ for all n then $\bigsqcup_n x_n \in I$.*

Using Lemma 2.5.1, one can easily show that there is a least ideal containing any given non-empty set of values, X. It is

$$Id(X) = \{x \mid \text{for all } \mathtt{n},\ [\![\mathtt{p}^{\mathtt{n}}_{\mathtt{n}}]\!](x) \sqsubseteq \text{some element of } X\}.$$

Generated ideals are ideals obtained from their denotable elements, as follows:

Definition 3.1.2. *I is a generated ideal if*

- *I is an ideal; and*
- *if $x \in I$ then there exists an increasing sequence of denotable values $\langle x_n \rangle$ such that $x_n \in I$ for all n and $x \sqsubseteq \bigsqcup x_n$.*

Analogously to the case of ideals, one has:

Theorem 3.1.3. *If X is a set of denotable elements, then $Id(X)$ is generated.*

Proof Certainly $Id(X)$ is an ideal. Suppose that x is an element of it. Set

$$R_n = \{z \in [\![\mathtt{p}^{\mathtt{n}}_{\mathtt{n}}]\!](\mathbf{V}) \mid z \text{ is denotable, } z \in Id(X), \text{ and } [\![\mathtt{p}^{\mathtt{n}}_{\mathtt{n}}]\!](x) \sqsubseteq z\}.$$

By Lemma 2.5.1 this set is finite. It is also non-empty as for any $\mathtt{n}$ there is a y in X such that $[\![\mathtt{p}^{\mathtt{n}}_{\mathtt{n}}]\!](x) \sqsubseteq y$. But then as y is denotable, and using Lemma 2.5.1 again, $[\![\mathtt{p}^{\mathtt{n}}_{\mathtt{n}}]\!](y)$ is in R_n. Finally, for every z' in R_{n+1} there is a z in R_n such that $z \sqsubseteq z'$ (take $z = [\![\mathtt{p}^{\mathtt{n}}_{\mathtt{n}}]\!](z')$ and use Lemma 2.5.1). We can therefore apply König's Lemma to the R_n to find an increasing sequence $\langle z_n \rangle$ (z_n in R_n), and using Lemma 2.5.1 one sees that this sequence verifies the second condition for $Id(X)$ being a generated ideal. □

It follows that the set of generated ideals, **GIdl**, is a complete lattice, with for any collection I_λ ($\lambda \in \Lambda$) of generated ideals:

$$\begin{aligned} \sqcup I_\lambda &= Id(\{x \in \cup I_\lambda \mid x \text{ is denotable}\}), \\ \sqcap I_\lambda &= Id(\{x \in \cap I_\lambda \mid x \text{ is denotable}\}). \end{aligned}$$

It is worth remarking that $I \sqcup J$ is $I \cup J$, that $\sqcup I_\lambda$ has the same finite denotable elements as $\cup I_\lambda$, although not, in general, the same denotable ones, and that $\sqcap I_\lambda$ has the same denotable elements as $\cap I_\lambda$.

Ideals are mathematically tractable in part because they are determined by their finite elements: indeed if I is an ideal then

$$I = Id(\{x \in I \mid x \text{ is finite}\}).$$

Analogously, generated ideals are mathematically tractable in part because they are determined by their finite denotable elements, as we show in Proposition 3.1.4 and Theorem 3.1.5. (But, not surprisingly, constructions with generated ideals sometimes require more work and care than their analogues with arbitrary ideals.)

Proposition 3.1.4. *Every denotable value is the limit of a chain of finite denotable values.*

Proof Every denotable element $[\![\mathtt{e}]\!]$ is the limit of an increasing sequence $\langle [\![\mathtt{p}^{\mathtt{n}}_{\mathtt{n}}(\mathtt{e})]\!] \rangle$ of finite denotable elements, by Lemma 2.5.1. □

Theorem 3.1.5. *Let I be a generated ideal. Then*

$$I = Id(\{x \in I \mid x \textit{ is finite and denotable}\}).$$

Proof In one direction, $I = Id(\{x \in I \mid x \text{ is finite}\}) \supseteq Id(\{x \in I \mid x \text{ is finite and denotable}\})$. In the other direction, by Proposition 3.1.4 every denotable element of I is in $Id(\{x \in I \mid x \text{ is finite and denotable}\})$, and then the second condition for I to be generated ensures that every element of I is in this set. □

3.2. Operations and Metric

In this subsection we define some operations and a metric on generated ideals. Most of the typical operations on ideals are unchanged for generated ideals; we define new versions of intersection and exponentiation. The metric is also new. Some symbols in the definitions below represent both functions on generated ideals within **V** and functions on domains such as **V**; the intended meaning of the symbols should be clear from context.

Definition 3.2.1. *Let I, J, J_1, ..., J_n be generated ideals, $\mathcal{K}$ a collection of generated ideals, and f a function from $GIdl^{n+1}$ to $GIdl$.*

- ***T*** *and* ***N*** *are the generated ideals of boolean values and natural numbers, respectively.*
- *$I \times J$ is the product of I and J.*
- *$I + J$ is the coalesced sum of I and J ($+$ identifies the least elements of its two arguments).*
- *$I \rightarrow J$ is a special sort of exponentiation of I and J: the smallest generated ideal that contains $\{[\![\boldsymbol{e}]\!] \in (\boldsymbol{V} \rightarrow \boldsymbol{V}) \mid$ if $[\![\boldsymbol{a}]\!] \in I$ then $[\![\boldsymbol{ea}]\!] \in J\}$.*
- *$(\forall_{\mathcal{K}} f)(J_1, \ldots, J_n) = \sqcap_{I \in \mathcal{K}} f(I, J_1, \ldots, J_n)$.*
- *$(\exists_{\mathcal{K}} f)(J_1, \ldots, J_n) = \sqcup_{I \in \mathcal{K}} f(I, J_1, \ldots, J_n)$.*
- *$(\mu f)(J_1, \ldots, J_n)$ is the unique I such that $I = f(I, J_1, \ldots, J_n)$, provided that f is contractive, in the sense defined below.*

It is simple to check that all the definitions are proper, with the exception of the definition of μ. That the definition of μ is proper follows from the Banach Fixpoint Theorem [8], stated below as Theorem 3.2.5. It is worth pointing out that our definition of $I \rightarrow J$ differs from the usual one even on denotable elements.

The usual metric on arbitrary ideals does not suit our purposes; in particular, we cannot exploit it to demonstrate the existence of solutions to recursive type equations. Generated ideals call for a different metric.

Definition 3.2.2. *Given two generated ideals I and J, their closeness $c(I, J)$ is the least rank of a denotable x such that $x \in I$ but $x \notin J$, or vice versa. We call such an x a witness for I and J. As usual, this determines a distance function: $d(I, J) = 2^{-c(I,J)}$.*

Theorem 3.2.3. *The generated ideals together with the distance function d form a complete metric space.*

Proof Theorem 3.1.5 guarantees that $c(I, J) = \infty$ if and only if $I = J$. In addition, it is obvious both that $c(I, J) = c(J, I)$ and that $c(I, K) \geq min(c(I, J), c(J, K))$. It remains to show that every Cauchy sequence converges. The proof of this is almost identical to the one for arbitrary ideals (Theorem 3 in [6]). Let $\langle I_n \rangle$ be a Cauchy sequence. Let I^o be the set of finite denotable members of almost all I_n. Then the limit of $\langle I_n \rangle$ is the generated ideal I that has the members of I^o as its denotable finite values. $\square$

As in the complete metric space defined by MacQueen, Plotkin, and Sethi, the notion of contractiveness and the Banach Fixpoint Theorem are the basic tools for proving that recursive type equations have unique solutions in this complete metric space.

Definition 3.2.4. *The function G is contractive if there exists a real number $0 \leq s < 1$ such that, for all $X_1, \ldots, X_n, X'_1, \ldots, X'_n$, we have*

$$d(G(X_1, \ldots, X_n), G(X'_1, \ldots, X'_n)) \leq s \cdot max\{d(X_i, X'_i) \mid 1 \leq n\}.$$

The function G is nonexpansive when $0 \leq s \leq 1$ instead.

Theorem 3.2.5 (.) *Banach If F is a contractive function on a complete metric space then the equation $X = F(X)$ has a unique solution.*

As could be expected, this unique solution is constructed by choosing an arbitrary X_0 and finding the limit of the sequence $X_0, F(X_0), F(F(X_0)), \ldots$.

In order to apply the Banach Fixpoint Theorem, it remains to prove that the operations defined above are contractive in the new metric, or at least nonexpansive in the new metric.

Theorem 3.2.6. *The operations $\sqcap$ and $\sqcup$ are nonexpansive. The operations $\times$, $+$, and $\rightarrow$ are contractive.*

Proof The general structure of the proof is taken from Proposition 6 and Theorem 7 of [6]. (A slightly more direct proof, which never mentions nondenotable elements, is easy to derive, but we prefer to imitate the previous proof in order to make similarities obvious.) Only $\rightarrow$ requires a new argument; we omit discussion of the other operations.

Let I, J, I', and J' be generated ideals. We want to show that $c(I \rightarrow J, I' \rightarrow J') > min(c(I, I'), c(J, J'))$. Consider a witness $[\![\mathtt{e}]\!]$ of least rank for the difference between $I \rightarrow J$ and $I' \rightarrow J'$.

Without loss of generality, we assume that $[\![\mathtt{ex}]\!] \in J$ for all $[\![\mathtt{x}]\!] \in I$, but $[\![\mathtt{ex}]\!] \notin J'$ for some $[\![\mathtt{x}]\!] \in I'$. In fact, we can even take this $[\![\mathtt{x}]\!]$ to be finite, since every denotable value is the limit of a chain of finite denotable values (by Proposition 3.1.4), application is continuous, and ideals are closed under approximations and limits.

Since $[\![\mathtt{e}]\!]$ must be a non-bottom function, Proposition 4 of [6] gives us $[\![\mathtt{e}]\!] = \bigsqcup(a_i \Rightarrow b_i)$ for some finite, non-zero number of a_i and b_i. Let $a = \bigsqcup\{a_i \mid a_i \sqsubseteq [\![\mathtt{x}]\!]\}$. Then $a \sqsubseteq [\![\mathtt{x}]\!]$ and $[\![\mathtt{e}]\!](a) = [\![\mathtt{ex}]\!] = \bigsqcup\{b_i \mid a_i \sqsubseteq [\![\mathtt{x}]\!]\}$. Immediately, a and $[\![\mathtt{ex}]\!]$ have rank smaller than $[\![\mathtt{e}]\!]$.

Two cases should be considered.

If $a \in I$ then we can take $[\![\mathtt{ex}]\!]$ as a witness for J and J'. To show this, we first observe that $[\![\mathtt{e}]\!](a) \in J$: a is less than the limit of a denotable chain of elements of I, which $[\![\mathtt{e}]\!]$ maps to denotable elements of J because $[\![\mathtt{e}]\!] \in I \rightarrow J$; then $[\![\mathtt{e}]\!](a) \in J$

by continuity and the definition of ideals. Since $[\![\mathtt{e}]\!](a) = [\![\mathtt{ex}]\!]$, we can take $[\![\mathtt{ex}]\!]$ as a witness for J and J'.

If $a \notin I$, we would hope to take it as a witness for I and I'. Unfortunately, a may not be denotable! On the other hand, $[\![\mathtt{x}]\!]$ is denotable, but its rank may be too large. We need to find a witness that combines the virtues of $[\![\mathtt{x}]\!]$ (denotability) and a (small rank).

To do this, we reduce the rank of $[\![\mathtt{x}]\!]$ with an application of a projection function: if $r(a) = \mathtt{n}$, we let $\mathtt{w} = \mathtt{p}^{\mathtt{m}}_{\mathtt{n}}(\mathtt{x})$, as described in Lemma 2.5.2.

Obviously, $[\![\mathtt{w}]\!]$ is denotable and its rank is no greater than the rank of a, which is less than the rank of $[\![\mathtt{e}]\!]$. Furthermore, $a \sqsubseteq [\![\mathtt{w}]\!]$. Hence $[\![\mathtt{w}]\!] \notin I$. On the other hand, $[\![\mathtt{w}]\!] \sqsubseteq [\![\mathtt{x}]\!]$ and hence $[\![\mathtt{w}]\!] \in I'$. Thus, $[\![\mathtt{w}]\!]$ is a denotable witness of smaller rank than $[\![\mathtt{e}]\!]$ for I and I'. □

Theorem 3.2.7. *If $\mathcal{K} \subseteq GIdl$ and f is contractive (nonexpansive) in its last n arguments then so are $(\forall_{\mathcal{K}} f)$ and $(\exists_{\mathcal{K}} f)$. If f is contractive (nonexpansive) then so is μf.*

Proof The arguments are identical to those for Theorems 8 and 9 in [6]. □

4. Semantics for Type Expressions

We can use the results on the complete metric space of generated ideals and on the operations in this space to give a semantics to a large class of type expressions. This class is the same one treated in [6] and suffices as the core of the type system for rich programming languages, such as Standard ML.

The generated-ideal semantics of type expressions is more accurate than the usual ones. Despite the features of generated ideals, however, the semantics is not quite faithful. Therefore, we restrict ourselves to subsets of the generated ideals. We define two subsets, the coarse ideals and the abstract ideals; they are rich enough for providing models for recursive polymorphic types. In both cases, the semantics of type expressions is faithful.

4.1. A Model Based on Generated Ideals

Often, one can determine that a function is contractive by examining the symbols used to express it. This motivates the definition of formally contractive type expressions—type expressions for functions that are "obviously" contractive. Roughly, formally contractive type expressions are those built up from **bool**, **int**, contractive operations, and nonexpansive operations followed by contractive operations.

Definition 4.1.1. *The expression σ is a formally contractive type expression in the variable t if one of the following conditions holds:*

- *σ has one of the forms **bool**, **int**, t' (with $t' \neq t$), $\sigma_1 \times \sigma_2$, $\sigma_1 + \sigma_2$, or $\sigma_1 \rightarrow \sigma_2$.*
- *σ has one of the forms $\sigma_1 \cap \sigma_2$ or $\sigma_1 \cup \sigma_2$, with both σ_1 and σ_2 formally contractive in t.*

- *σ has one of the forms $\forall t'.\sigma_1$, $\exists t'.\sigma_1$, or $\mu t'.\sigma_1$, with either $t = t'$ or σ_1 formally contractive in t.*

The definition of formally contractive type expressions naturally leads to the definition of well-formed type expressions:

Definition 4.1.2. *The expression σ is a well-formed type expression if one of the following conditions holds:*

- *σ is **bool**, **int**, or t.*
- *σ has one of the forms $\sigma_1 \times \sigma_2$, $\sigma_1 + \sigma_2$, $\sigma_1 \rightarrow \sigma_2$, $\sigma_1 \cap \sigma_2$, or $\sigma_1 \cup \sigma_2$, with both σ_1 and σ_2 well-formed.*
- *σ has one of the forms $\forall t.\sigma_1$ or $\exists t.\sigma_1$, with σ_1 well-formed.*
- *σ has the form $\mu t.\sigma_1$, with σ_1 well-formed and formally contractive in t.*

TEXP *is the set of well-formed type expressions.*

$$[\![\ ]\!] : \mathrm{TEXP} \rightarrow (\mathrm{TVAR} \rightarrow \mathit{Idl}) \rightarrow \mathit{Idl}$$

$$\begin{array}{lcl}
[\![\mathtt{bool}]\!]_\rho & = & \mathbf{T} \\
[\![\mathtt{int}]\!]_\rho & = & \mathbf{N} \\
[\![t]\!]_\rho & = & \rho(t) \\
[\![\sigma_1 \cap \sigma_2]\!]_\rho & = & [\![\sigma_1]\!]_\rho \sqcap [\![\sigma_2]\!]_\rho \\
[\![\sigma_1 \cup \sigma_2]\!]_\rho & = & [\![\sigma_1]\!]_\rho \sqcup [\![\sigma_2]\!]_\rho \\
[\![\sigma_1 \times \sigma_2]\!]_\rho & = & [\![\sigma_1]\!]_\rho \times [\![\sigma_2]\!]_\rho \\
[\![\sigma_1 + \sigma_2]\!]_\rho & = & [\![\sigma_1]\!]_\rho + [\![\sigma_2]\!]_\rho \\
[\![\sigma_1 \rightarrow \sigma_2]\!]_\rho & = & [\![\sigma_1]\!]_\rho \rightarrow [\![\sigma_2]\!]_\rho \\
[\![\forall t.\sigma]\!]_\rho & = & \forall_{\mathcal{K}}(\lambda I \in \mathit{Idl}.[\![\sigma]\!]_{\rho\{t \leftarrow I\}}) \\
[\![\exists t.\sigma]\!]_\rho & = & \exists_{\mathcal{K}}(\lambda I \in \mathit{Idl}.[\![\sigma]\!]_{\rho\{t \leftarrow I\}}) \\
[\![\mu t.\sigma]\!]_\rho & = & \mu(\lambda I \in \mathit{Idl}.[\![\sigma]\!]_{\rho\{t \leftarrow I\}})
\end{array}$$

Fig. 2: Definition of the meaning function for type expressions

Imitating MacQueen, Plotkin, and Sethi, we can define a semantics for well-formed type expressions. The meaning function $[\![\ ]\!]^{\mathrm{G}}$ associates a generated ideal with each well-formed type expression under each type assignment for the free type variables in the expression: if TVAR is the set of type variables, then

$$[\![\ ]\!]^{\mathrm{G}} : \mathrm{TEXP} \rightarrow (\mathrm{TVAR} \rightarrow \mathit{GIdl}) \rightarrow \mathit{GIdl}.$$

We define the semantics in Fig. 2. There, we use **Idl** to refer to the collection of ideals under consideration, in this case **GIdl**, and $\mathcal{K}$ to refer to $\{I \in \mathit{Idl} \mid \mathtt{wrong} \notin I\}$ (the same semantic templet reappears below, and **Idl** refers to other sets of ideals).

We can prove:

Theorem 4.1.3. The generated-ideal semantics is well defined.

Proof The proof is straightforward, given the machinery of the previous section. It resembles the proof of Theorem 10 in [6]. □

What is the type of the `explode-||-or` function, according to this semantics? Whenever $[\![$`explode-||-or`$]\!]$ is given a denotable argument in the generated ideal $\mathbf{T}\times\mathbf{T}\rightarrow\mathbf{T}$, it yields a result in $\mathbf{T}$. Thus, `explode-||-or` is of type $(bool\times bool\rightarrow bool)\rightarrow bool$, as we originally suggested.

We may also notice that the expression

```
ignore-||-or = λ x.
                  if  x(true, btm)
                  and x(btm, true)
                  and not x(false, false)
                  then true
    else true
```

behaves identically to `explode-||-or` in all contexts, and, as we would desire, also has type $(bool\times bool\rightarrow bool)\rightarrow bool$.

These observations indicate that the semantics fits reasonably well with the programming language. The fit is also confirmed by the remarks in Section 5 on the isomorphism between generated ideals and sets of terms.

However, the fit is not as close as it could be. The generated ideals in $\mathbf{V}$ may distinguish two expressions with identical behavior. As a trivial example,

$$[\![\texttt{explode-||-or}]\!] \in \{v \mid v \sqsubseteq [\![\texttt{explode-||-or}]\!]\}$$

while

$$[\![\texttt{ignore-||-or}]\!] \notin \{v \mid v \sqsubseteq [\![\texttt{explode-||-or}]\!]\}.$$

Since a free type variable can take $\{v \mid v \sqsubseteq [\![\texttt{explode-||-or}]\!]\}$ as meaning, the generated-ideal semantics fails to be faithful. (It is an open question whether the generated-ideal semantics is faithful when restricted to type expressions with no free type variables.) In order to guarantee a perfect fit between syntax and semantics, a more finely tuned concept of type is needed.

4.2. Two Faithful Models

In this subsection we consider concepts of type rich enough as a basis to a semantics for recursive polymorphic types, yet coarse enough not to make irrelevant distinctions between expressions that behave identically. The crucial observation is that we do not need all of **GIdl** to define a semantics. We restrict our attention to two subsets of **GIdl**, the coarse ideals, **CIdl**, and the abstract ideals, **AIdl**.

The subset **CIdl** is close to the minimum needed for a suitable semantics for recursive polymorphic types. For instance, $\mathbf{T}$ is in **CIdl** while (we conjecture) $\{\perp, 3\}$ is not; this seems acceptable, since $\mathbf{T}$ is useful as the meaning of **bool**, while $\{\perp, 3\}$ is of no use at all as a programming type.

Definition 4.2.1. *CIdl is the smallest subset of GIdl that contains* $\boldsymbol{T}$ *and* $\boldsymbol{N}$*, and is closed under binary and infinitary* $\sqcap$ *and* $\sqcup$*, under* $\times$*,* $+$*, and* $\rightarrow$*, and under limits of Cauchy sequences.*

The closure conditions on the set of ideals **CIdl** are monotone (with respect to set inclusion), hence Definition 4.2.1 is proper.

We can reproduce the semantics of type expressions using coarse ideals instead of generated ideals. We define

$$[\![\ ]\!]^{\mathrm{C}} : \mathrm{TE}\textsc{xp} \rightarrow (\mathrm{TV}\textsc{ar} \rightarrow CIdl) \rightarrow CIdl.$$

The semantics is given by Fig. 2 (with **CIdl** playing the role of **Idl**). We apply the Banach Fixpoint Theorem as usual, to prove that the semantics is well defined.

Theorem 4.2.2. *The coarse-ideal semantics is well defined.*

Proof Again, the proof is straightforward, given the machinery of the previous section. It resembles the proof of Theorem 10 in [6]. □

It remains to show that this semantics is faithful. For this purpose, we use **AIdl**, the class of abstract generated ideals—for short, abstract ideals. (Abstract sets were defined in Subsection 2.4.)

Definition 4.2.3. ***AIdl*** *is the set of abstract generated ideals.*

Abstract ideals serve in the study of coarse ideals, and also give rise to a natural semantics of type expressions, of independent interest. It turns out that **CIdl** is a subset of **AIdl**. Hence, a semantics based on **CIdl** benefits from the features of **AIdl**.

To guarantee that **CIdl** and **AIdl** both suffice as the basis for a faithful semantics, we prove that they share each other's features. **AIdl** enjoys the same closure properties as **CIdl**. Hence **CIdl**, the smallest set with these closure properties, must be a subset of **AIdl**. In other words, all coarse ideals are abstract (and probably not vice versa: $\{\perp, 3\}$ is abstract but apparently not coarse).

Lemma 4.2.4. ***AIdl*** *contains* ***T*** *and* ***N****, and is closed under binary and infinitary $\sqcap$ and $\sqcup$, under $\times$, $+$, and $\rightarrow$, and under limits of Cauchy sequences. If f is a function from $AIdl^{n+1}$ to* ***AIdl*** *then so are $(\forall_{\mathcal{K}} f)$, $(\exists_{\mathcal{K}} f)$, and, provided f is contractive, (μf).*

Proof We consider each of the closure conditions in turn. We assume that the arguments (if any) to an operation are abstract, to prove that the result is abstract as well. The cases vary in difficulty.

- The proposition is trivial for **T**. Suppose that `e'` $\underset{\sim}{\sqsubseteq}$ `e`, that is, `e'` yields $\perp$ in all contexts where `e` does. This must be true, in particular, for the context `if (_ or not _) then btm else wrong`, hence if $[\![\mathtt{e}]\!] \in \mathbf{T}$ then $[\![\mathtt{e'}]\!] \in \mathbf{T}$.

- The proposition is also trivial for **N**. Suppose that `e'` $\underset{\sim}{\sqsubseteq}$ `e`, that is, `e'` yields $\perp$ in all contexts where `e` does. This must be true, in particular, for the context `if (z(_) or not z(_)) then btm else wrong`, and hence if $[\![\mathtt{e}]\!] \in \mathbf{N}$ then $[\![\mathtt{e'}]\!] \in \mathbf{N}$.

- The cases of binary $\sqcap$ and $\sqcup$, infinitary $\sqcap$, and $\times$ and $+$ are solved with simple applications of the hypothesis. (When considering binary $\sqcap$ and $\sqcup$ and infinitary $\sqcap$, one should recall that they coincide with the corresponding set-theoretic operations on denotable elements.)

- For $\rightarrow$, suppose that I and J are abstract, and that $\mathtt{e'} \precsim \mathtt{e}$ and $[\![\mathtt{e}]\!] \in (I \rightarrow J)$. The definition of $\precsim$ yields that $\mathtt{e'(a)} \precsim \mathtt{e(a)}$, for all $\mathtt{a}$. Furthermore, if $[\![\mathtt{a}]\!] \in I$ then $[\![\mathtt{e(a)}]\!] \in J$ and, since J is abstract, $[\![\mathtt{e'(a)}]\!] \in J$. By the definition of $\rightarrow$, it follows that $[\![\mathtt{e'}]\!] \in (I \rightarrow J)$, as we wanted to show.

- The case for infinitary $\sqcup$ requires a slightly more elaborate argument. Assume that $\mathtt{e'} \precsim \mathtt{e}$. We consider chains $\langle [\![\mathtt{p}^{\mathtt{n}}_{\mathtt{n}}\mathtt{(e)}]\!] \rangle$ and $\langle [\![\mathtt{p}^{\mathtt{n}}_{\mathtt{n}}\mathtt{(e')}]\!] \rangle$ of finite denotable elements with limits $[\![\mathtt{e}]\!]$ and $[\![\mathtt{e'}]\!]$, respectively (see Proposition 3.1.4). The definition of $\precsim$ yields $\mathtt{p}^{\mathtt{n}}_{\mathtt{n}}\mathtt{(e')} \precsim \mathtt{p}^{\mathtt{n}}_{\mathtt{n}}\mathtt{(e)}$ for all $\mathtt{n}$. Since each of the ideals coming into the union is abstract by the hypothesis, for all $\mathtt{n}$, each ideal must contain $[\![\mathtt{p}^{\mathtt{n}}_{\mathtt{n}}\mathtt{(e')}]\!]$ if it contains $[\![\mathtt{p}^{\mathtt{n}}_{\mathtt{n}}\mathtt{(e)}]\!]$. If $[\![\mathtt{e}]\!]$ is a limit point of the union of the ideals then $[\![\mathtt{p}^{\mathtt{n}}_{\mathtt{n}}\mathtt{(e)}]\!]$ is in the union, for all $\mathtt{n}$. Then $[\![\mathtt{p}^{\mathtt{n}}_{\mathtt{n}}\mathtt{(e')}]\!]$ is in the union too, for all $\mathtt{n}$, and finally $[\![\mathtt{e'}]\!]$ is a limit point of the union as well.

- Closure under limits of Cauchy sequences requires an argument similar to the previous one. Assume that $\mathtt{e'} \precsim \mathtt{e}$. We consider chains $\langle [\![\mathtt{p}^{\mathtt{n}}_{\mathtt{n}}\mathtt{(e)}]\!] \rangle$ and $\langle [\![\mathtt{p}^{\mathtt{n}}_{\mathtt{n}}\mathtt{(e')}]\!] \rangle$ of finite denotable elements with limits $[\![\mathtt{e}]\!]$ and $[\![\mathtt{e'}]\!]$, respectively (see Proposition 3.1.4). The definition of $\precsim$ yields $\mathtt{p}^{\mathtt{n}}_{\mathtt{n}}\mathtt{(e')} \precsim \mathtt{p}^{\mathtt{n}}_{\mathtt{n}}\mathtt{(e)}$ for all $\mathtt{n}$. Since each of the ideals in the sequence is abstract by the hypothesis, for all $\mathtt{n}$, each ideal must contain $[\![\mathtt{p}^{\mathtt{n}}_{\mathtt{n}}\mathtt{(e')}]\!]$ if it contains $[\![\mathtt{p}^{\mathtt{n}}_{\mathtt{n}}\mathtt{(e)}]\!]$. If the limit contains $[\![\mathtt{e}]\!]$ then it must contain $[\![\mathtt{p}^{\mathtt{n}}_{\mathtt{n}}\mathtt{(e)}]\!]$. Therefore, since the sequence is a Cauchy sequence, all sufficiently late terms in the sequence contain $[\![\mathtt{p}^{\mathtt{n}}_{\mathtt{n}}\mathtt{(e)}]\!]$ (only finitely many terms can differ from the limit on elements of rank $\mathtt{n}$). In turn, it follows that all sufficiently late terms in the sequence must also contain $[\![\mathtt{p}^{\mathtt{n}}_{\mathtt{n}}\mathtt{(e')}]\!]$, and then the limit of the sequence must contain $[\![\mathtt{p}^{\mathtt{n}}_{\mathtt{n}}\mathtt{(e')}]\!]$. Finally, the limit of the sequence must also contain $[\![\mathtt{e'}]\!]$.

- Similarly, if f is contractive and maps abstract ideals to abstract ideals then so do $(\forall_{\mathcal{K}} f)$ and $(\exists_{\mathcal{K}} f)$: this follows immediately from their definition, together with the closure of **AIdl** under infinitary $\sqcap$ and $\sqcup$.

- If f maps abstract ideals to abstract ideals then so does (μf): this follows immediately from the definition of μ, together with the closure of **AIdl** under limits of Cauchy sequences (since fixpoints are obtained as limits of Cauchy sequences). □

To finish, we define an abstract-ideal semantics and prove that both the coarse-ideal semantics and the abstract-ideal semantics are faithful.

We define the abstract-ideal semantics using the templet of Fig. 2 (with **AIdl** playing the role of **Idl**):

$$[\![\ ]\!]^{\mathsf{A}} : \mathrm{TExp} \rightarrow (\mathrm{TVar} \rightarrow AIdl) \rightarrow AIdl.$$

Theorem 4.2.5. *The abstract-ideal semantics is well defined.*

Proof First we should point out that **AIdl** is closed under the necessary type operations, as shown in Lemma 4.2.4. Then the argument is the usual one. □

Theorem 4.2.6. *Both the coarse-ideal semantics and the abstract-ideal semantics are faithful.*

Proof Both semantics map type expressions to elements of **AIdl**, which are all abstract by definition. (For the coarse-ideal semantics, this follows from Lemma 4.2.4.) □

This result is fairly robust with respect to changes in the interpretation of terms. Consider an interpretation more abstract than ours, that is, an interpretation that identifies the meanings of more programs. If the interpretation is sound, some meanings of programs will be identified within abstract sets, but not across the boundaries of abstract sets. Therefore, we can map the abstract ideals of our semantics for type expressions to ideals over the more abstract model, and obtain the corresponding faithful semantics.

5. Syntactic Solutions

While generated ideals have close connections with the world of expressions, they still belong to the world of values. In some situations, a more syntactic model, where types are certain sets of expressions, may seem desirable. (In fact, this work originated in just such a situation; the preference for a syntactic model was simply a matter of taste.)

In this section we take advantage of the properties of generated ideals to discuss models of recursive polymorphic types where sets of expressions are used instead of sets of values. Finally, we speculate on a totally syntactic model.

5.1. Term Ideals

A simple way to obtain a model where types are sets of expressions is to induce these sets from the abstract ideals.

Definition 5.1.1. *If $I \subseteq \boldsymbol{V}$ then the representation of I is the set of all closed expressions e such that $[\![\mathsf{e}]\!] \in I$.*

Definition 5.1.2. *I is a term ideal if it is the representation of some abstract ideal. **TIdl** is the set of term ideals, and R is the one-to-one function that maps abstract ideals to their representations.*

Immediately, we can define a semantics of type expression using term ideals,

$$[\![\ ]\!]^{\mathrm{T}} : \mathrm{TE{\scriptstyle XP}} \rightarrow (\mathrm{TV{\scriptstyle AR}} \rightarrow TIdl) \rightarrow TIdl,$$

thus

$$[\![\sigma]\!]^{\mathrm{T}}_{\rho} = R([\![\sigma]\!]^{\wedge}_{R^{-1}\circ\rho}).$$

Actually, the term ideals can be characterized directly, rather naturally.

Theorem 5.1.3. *I is a term ideal if and only if*

- *$I \neq \emptyset$; and*
- *for all sequences $\mathtt{e}_1 \precsim \mathtt{e}_2 \precsim \ldots$ of elements of I, for all $\mathtt{e}$, if for all $\mathcal{C}$ such that $\mathtt{e} \models \mathcal{C}$ there exists i such that $\mathtt{e}_\mathtt{i} \models \mathcal{C}$, then $\mathtt{e} \in I$.*

Proof Assume that I is a term ideal, that $\mathtt{e}_1 \precsim \mathtt{e}_2 \precsim \ldots$ is a sequence of elements of I, and that for all $\mathcal{C}$ if $\mathtt{e} \models \mathcal{C}$ then $\mathtt{e}_\mathtt{i} \models \mathcal{C}$ for some $\mathtt{i}$. We wish to show that $\mathtt{e} \in I$. Trivially, it suffices to argue that each $\mathtt{p}^\mathtt{n}_\mathtt{n}\mathtt{(e)} \in I$. As the range of $[\![\mathtt{p}^\mathtt{n}_\mathtt{n}]\!]$ is finite, the sequence $\langle \mathtt{p}^\mathtt{n}_\mathtt{n}\mathtt{(e}_\mathtt{i}\mathtt{)} \rangle$ contains infinitely many times the same element (up to semantic equality). Let $\mathtt{p}^\mathtt{n}_\mathtt{n}\mathtt{(e}_\mathtt{m}\mathtt{)}$ be such an element. Suppose that $\mathtt{p}^\mathtt{n}_\mathtt{n}\mathtt{(e)} \models \mathcal{C}$. Then there exists $\mathtt{k}$ such that $\mathtt{p}^\mathtt{n}_\mathtt{n}\mathtt{(e}_\mathtt{k}\mathtt{)} \models \mathcal{C}$, and the assumption that $\mathtt{e}_1 \precsim \mathtt{e}_2 \precsim \ldots$ yields $\mathtt{p}^\mathtt{n}_\mathtt{n}\mathtt{(e}_\mathtt{j}\mathtt{)} \models \mathcal{C}$ for all $\mathtt{j} \geq \mathtt{k}$. In particular, $\mathtt{p}^\mathtt{n}_\mathtt{n}\mathtt{(e}_\mathtt{m}\mathtt{)} \models \mathcal{C}$, since $\mathtt{p}^\mathtt{n}_\mathtt{n}\mathtt{(e}_\mathtt{m}\mathtt{)}$ arises infinitely often in the sequence $\langle \mathtt{p}^\mathtt{n}_\mathtt{n}\mathtt{(e}_\mathtt{i}\mathtt{)} \rangle$. Then we have $\mathtt{p}^\mathtt{n}_\mathtt{n}\mathtt{(e)} \precsim \mathtt{p}^\mathtt{n}_\mathtt{n}\mathtt{(e}_\mathtt{m}\mathtt{)} \precsim \mathtt{e}_\mathtt{m} \in I$ and, therefore, $\mathtt{p}^\mathtt{n}_\mathtt{n}\mathtt{(e)} \in I$.

Conversely, suppose that I is a non-empty set that satisfies the given closure condition. We have to find an abstract ideal J such that $I = R(J)$. Let

$$J = Id(\{[\![\mathtt{e}]\!] \mid \mathtt{e} \in I\}).$$

By Theorem 3.1.3 this is a generated ideal.

Evidently, we have $I \subseteq R(J)$. In order to show the converse, suppose that $[\![\mathtt{e'}]\!]$ is in $R(J)$. Then for every $\mathtt{n}$, $[\![\mathtt{p}^\mathtt{n}_\mathtt{n}\mathtt{(e')}]\!] \sqsubseteq [\![\mathtt{e}]\!]$ for some $\mathtt{e} \in I$. The closure condition, applied to the constant sequence $\mathtt{e} \precsim \mathtt{e} \precsim \ldots$, yields $\mathtt{p}^\mathtt{n}_\mathtt{n}\mathtt{(e')} \in I$. Finally, we consider the sequence $\mathtt{p}^1_1\mathtt{(e')} \precsim \mathtt{p}^2_2\mathtt{(e')} \precsim \ldots$, and conclude that $\mathtt{e'} \in I$.

As for abstraction, suppose $\mathtt{e'} \precsim \mathtt{e}$ and $[\![\mathtt{e}]\!] \in J$; again, we consider the constant sequence $\mathtt{e} \precsim \mathtt{e} \precsim \ldots$, to derive that $\mathtt{e'} \in I$ and, therefore, that $[\![\mathtt{e'}]\!] \in J$. □

The proof of Theorem 5.1.3 suggests a useful formula for the inverse of R:

$$R^{-1}(I) = Id(\{[\![\mathtt{e}]\!] \mid \mathtt{e} \in I\}).$$

The operations on term ideals have direct definitions, and for example we can write

$$I \sqcap J = I \cap J$$

and

$$I \rightarrow J = \{\mathtt{e} \mid [\![\mathtt{e}]\!] \in (\mathbf{V} \rightarrow \mathbf{V}) \text{ and } \forall \mathtt{e'} \in I.\, \mathtt{e(e')} \in J\}.$$

These definitions correspond with those for **AIdl**, in the sense that, for example

$$I \sqcap J = R(R^{-1}(I) \cap R^{-1}(J))$$

and

$$I \rightarrow J = R(R^{-1}(I) \rightarrow R^{-1}(J)).$$

With their aid we can easily cast the definition of $[\![\ ]\!]^\mathrm{T}$ in a homomorphic form, as in Fig. 2.

5.2. *Operational Ideals*

We have not defined a reduction relation for expressions in the programming language, or for that matter any kind of operational semantics for the programming language. To finish, we speculate on an alternative, even more syntactic development of our theory. This development requires the use of a well-behaved reduction relation. (However, we do not propose a particular reduction relation.)

We imagine that we have distinguished a set of "canonical" closed expressions. For example, the canonical boolean expressions are `true` and `false`. Similarly, some set F of canonical expressions are distinguished as functions. We write $\mathtt{e} \Rightarrow \mathtt{e'}$ for "the closed expression `e` reduces to the canonical expression `e'`."

An operational definition for $\mathbf{T}$ seems reasonable in this setting:

$$\mathbf{T} \ = \{\mathtt{e} \mid \text{ if } \mathtt{e} \Rightarrow \mathtt{e'} \text{ then } \mathtt{e'} \in \{\mathtt{true}, \mathtt{false}\}\}.$$

We may define function types similarly:

$$I \rightarrow J = \{\mathtt{e} \mid \text{ if } \mathtt{e} \Rightarrow \mathtt{e'} \text{ then } \mathtt{e'} \in \mathrm{F} \text{ and if } \mathtt{a} \in I \text{ then } \mathtt{ea} \in J\}.$$

For suitable notions of reduction, these definitions should coincide with the previous ones. Proving that these definitions do coincide, however, may pose a hard adequacy problem; see [4, 9] for a discussion of adequacy. For instance, one has to show that $[\![\mathtt{e}]\!] \in \mathbf{T}$ if and only if $\mathtt{e} \Rightarrow \mathtt{e'}$ implies $\mathtt{e'} \in \{\mathtt{true}, \mathtt{false}\}$, and similarly that $[\![\mathtt{e}]\!] \in (\mathbf{V} \rightarrow \mathbf{V})$ if and only if $\mathtt{e} \Rightarrow \mathtt{e'}$ implies $\mathtt{e'} \in \mathbf{F}$.

In the end, we obtain a class of "operational ideals." This approach, though laborious, provides totally syntactic solutions to recursive type equations.

6. Conclusions

Recursive polymorphic types can be modeled with various degrees of accuracy. MacQueen, Plotkin, and Sethi have defined a pure ideal semantics, with no reference to the syntax of the programming language considered; some type-inference rules for the language are semantically sound, but the correspondence between syntax and semantics goes not much further. We have introduced some simple syntactic notions into our ideal semantics, obtaining first the generated ideals and then the coarse ideals and the abstract ideals. The coarse-ideal semantics and the abstract-ideal semantics are slightly more complex than the original ideal semantics, but as accurate as possible, that is, faithful.

It remains unclear how to extend our methods to other type systems, for example to systems with explicit polymorphism. It seems clear, however, that faithfulness is a generally relevant issue in the semantics of type systems.

Acknowledgements

We are grateful to Luca Cardelli, Stavros Cosmadakis, and an anonymous referee for useful suggestions, and to Cynthia Hibbard for editorial help.

References

[1] Robin Milner. Fully abstract models of typed λ-calculi. *Theoretical Computer Science*, 4:1–22, 1977.

[2] Gordon Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–256, 1977.

[3] Allen Stoughton. *Fully Abstract Models of Programming Languages*. Pitman/Wiley, 1988. Research Notes in Theoretical Computer Science.

[4] Albert Meyer. Semantical paradigms: notes for an invited lecture. In *Proceedings of the Third Annual IEEE Symposium on Logic in Computer Science*, pages 236–253, July 1988. With two appendices by Stavros S. Cosmadakis.

[5] Robin Milner. A proposal for Standard ML. In *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 184–197, August 1984.

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

[7] Henk P. Barendregt. *The Lambda Calculus*. North Holland, Revised edition, 1984.

[8] Stefan Banach. Sur les opérations dans les ensembles abstraits et leurs applications aux équations intégrales. *Fund. Math.*, 3:7–33, 1922.

[9] Stavros S. Cosmadakis. Computing with recursive types. In *Proceedings of the Fourth Annual IEEE Symposium on Logic in Computer Science*, pages 24–38, June 1989.