Parametricity and Proving Free Theorems for Functional-Logic Languages

Stefan Mehner, Daniel Seidel, Lutz Strassburger, Janis Voigtländer

Parametricity and Proving Free Theorems for Functional-Logic Languages

Stefan Mehner *
Institut für Informatik
Universität Bonn
mehner@cs.uni-bonn.de

Daniel Seidel†
Institut für Informatik
Universität Bonn
ds@cs.uni-bonn.de

Lutz Straßburger
INRIA & LIX
Ecole Polytechnique
lutz@lix.polytechnique.fr

Janis Voigtländer
Institut für Informatik
Universität Bonn
jv@cs.uni-bonn.de

Abstract

The goal of this paper is to provide the required foundations for establishing free theorems – statements about program equivalence, guaranteed by polymorphic types – for the functional-logic programming language Curry. For the sake of presentation we restrict ourselves to a language fragment that we call CuMin, and that has the characteristic features of Curry (both functional and logic). We present a new denotational semantics based on partially ordered sets without limits. We then introduce an intermediate language called SaLT that is essentially a lambda-calculus extended with an abstract set type, and again give a denotational semantics. We show that the standard (logical relations) techniques can be applied to obtain a general parametricity theorem for SaLT and derive free theorems from it. Via a translation from CuMin to SaLT that fits the respective semantics, we then derive free theorems for CuMin.

Categories and Subject Descriptors F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.1.6 [*Programming Techniques*]: Logic Programming; D.3.1 [*Programming Languages*]: Formal Definitions and Theory—Semantics; D.3.2 [*Programming Languages*]: Language Classifications—Multiparadigm languages; D.3.3 [*Programming Languages*]: Language Constructs and Features—Polymorphism, Recursion; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Denotational semantics; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Functional constructs, Type structure

1. Introduction

One virtue of logic programming languages is their being intrinsically nondeterministic. Functional programming languages are able to emulate this behavior by using lists in order to keep track of the many possibilities. But the price is that one has to deface the syntax by additional combinators to handle those lists. Functional-logic languages like TOY [12] and Curry [10] combine both paradigms, thus enhancing functional programming through logical features like free variables and choice.

*,† The first and second author were supported by the DFG under grant VO 1512/1-2.

PPDP '14, September 8–10, 2014, Canterbury, UK.

ACM 978-1-4503-2947-7/14/09…$15.00.
http://dx.doi.org/10.1145/2643135.2643147

At the same time, TOY and Curry inherit strong polymorphic type systems from their functional background. That brings potential benefits for program analysis, transformation, and performance optimization. A popular type-based reasoning method for establishing semantic properties of and in typed functional languages like Haskell are *free theorems* [17], which are successfully applied in a multitude of ways in the functional programming world [8, 16, and many others]. A simple example of a free theorem is the following: For every polymorphic function $f :: [\alpha] \to [\alpha]$ from lists to lists, arbitrary types τ_1 and τ_2, and a function $g :: \tau_1 \to \tau_2$, we have

$$f \circ (map\ g) = (map\ g) \circ f$$

for the standard function $map :: (\alpha \to \beta) \to [\alpha] \to [\beta]$ that takes a function and a list and applies that function to every entry of the list. This can be useful, e.g., if one has a "pipeline" $(map\ h) \circ f \circ (map\ g)$ in a program, which one can now replace by $(map\ h) \circ (map\ g) \circ f$ and hence by $(map\ (h \circ g)) \circ f$, while in the original expression h and g are not in reach of each other for being fused. The important aspect here is that f can be arbitrary (*reverse*, *tail*, …), only its type being $[\alpha] \to [\alpha]$ is relevant[1] – that is the sense in which free theorems are *free* (while there may well be conditions on g, depending on the specific language setting).

Can we bring the powerful tool of free theorems to functional-logic languages? As a matter of fact, the example equation $f \circ (map\ g) = (map\ g) \circ f$ for polymorphic f does *not* hold with the same generality there. But what does? Previous work [4] has investigated free theorems for Curry phenomenologically and provides intuition for required premises of free theorems (such as conditions on g in the example above) as well as counterexamples (if said conditions are violated). Proof of the positive claims (such as validity of the free theorem for suitably conditioned g) has been elusive so far, largely because Curry's (as well as TOY's) type system conceals usage of the key feature: nondeterminism. This is convenient for programmers, as they do not have to distinguish between deterministic and nondeterministic values. However, it is a hindrance to formal reasoning: the conditions identified in said work include a notion of (weakened) determinism, and hence a type system capturing this concept would be preferable.

Let be given a typed language with polymorphic types. In order to establish free theorems, one first has to prove a general parametricity theorem [15]. Roughly speaking, parametricity establishes a relation between two different semantic interpretations of an expression. From this, free theorems can be derived by specializing relations to (the semantics of) functions. A nondeterministic language needs some kind of set- (or multiset-) valued semantics in

[1] That polymorphic type means that f can operate only on the structure of its argument, not on the elements therein. That restricts what it can do, thus giving rise to the above equation. This would fail for a function not so polymorphic, for example a function of type $[\mathsf{Int}] \to [\mathsf{Int}]$ squaring all input list numbers.

order to accommodate the ambiguity of evaluation results. Such a semantics has been given for a suitable sublanguage of Curry in [5], but no parametricity theorem was proved in this "lifted" setting. Moreover, even if one were proved, the usual machinery for deriving free theorems would not be available anymore, because the relations occurring in the parametricity theorem would no longer be compatible with the graphs of the *set-valued functions* provided by the semantics of the programming language.

To overcome these difficulties, this paper proceeds in the following steps. In Sections 2 and 3, we describe a representative sublanguage of Curry, called CuMin (for *Curry Minor*), and establish a denotational semantics for it, which is based on pointed partially ordered sets and can be seen as a simplification of the denotational semantics given in [5, 6] based on directed complete partial orders and used as a basis for further semantic investigations into Curry by [7]. Next, we define an intermediate language called SaLT (for *Set- and Lambda-Terms*) which is a dedicated lambda-calculus with explicit set types and monadic operations on those, along with its semantics (Section 4). We also define a purely syntactic translation from CuMin programs to SaLT programs that preserves the semantics (Section 5). This allows us to express, within the SaLT type system, where in the original CuMin program nondeterminism is or is not actually used (Section 6). Moreover, we can apply the standard techniques to prove a parametricity theorem for SaLT and can derive free theorems from it (Section 7). Finally, we can use our semantics and translation to give formal definitions to the informal conditions from [4] and derive free theorems for CuMin from the corresponding free theorems for SaLT (Section 8). In the end, we give four examples of free theorems for CuMin that are derived using our method (Section 9).

Parametricity in the presence of nondeterminism has been considered before, e.g., as a special case of a general framework for computational effects in [14]. To our knowledge, such studies have included a choice operator, but not logical free variables (which we show bring extra constraints into the picture). In fact, it is possible that SaLT without the logical feature of free variables could be expressed by choosing an appropriate monad instance in the setup of [14], provided the latter were extended to cover general recursion. But even then, and even if the monad were successfully enriched to also cover logical free variables, this would not directly result in free theorems for existing functional-logic languages like TOY and Curry. We are not interested here in lambda-calculi with nondeterminism features per se, but as a means to establish reasoning tools for those languages. This does not permit a clean slate approach, rather we have to take the specific semantic intricacies of the existing languages (like call-time choice [11]) into account. That explains why coming up with the right semantic setup and appropriate translations forward (of programs) and backward (of free theorems) is an as important part of our development, besides the parametricity theorem for SaLT itself.

2. The Language CuMin

We introduce a sublanguage *CuMin* of Curry [10]. It is related to FlatCurry, which is a middle-end representation used for example in the PAKCS compiler and also in semantic investigations in the literature [2].

The CuMin syntax is shown in Figure 1, where $\overline{x_n}$ represents n variables $x_1 \ldots x_n$, and $\overline{\tau_m}$ accordingly. Most Curry can be expressed in CuMin[2], though the limited syntax requires some code transformations to eliminate where-clauses, lambda-abstractions, pattern-matching on left-hand sides of function definitions, guards and such. The resulting adjusted code is a sequence of top-level

[2] A notable exception is local recursion à la letrec. See also the next footnote further below.

$$
\begin{array}{rcl}
P & ::= & D; P \mid D \\
D & ::= & f :: \kappa\,\tau;\ f\ \overline{x_n} = e \\
\kappa & ::= & \forall^{\varepsilon}\alpha.\kappa \mid \forall^{*}\alpha.\kappa \mid \varepsilon \\
\tau & ::= & \alpha \mid \mathsf{Bool} \mid \mathsf{Nat} \mid [\tau] \mid (\tau,\tau') \mid \tau \to \tau' \\
e & ::= & x \mid f_{\overline{\tau_m}} \mid e_1\ e_2 \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \mid n \mid e_1 + e_2 \mid e_1 \equiv e_2 \\
 & & \mid (e_1, e_2) \mid \mathsf{case}\ e\ \mathsf{of}\ \langle (x,y) \to e_1 \rangle \\
 & & \mid \mathsf{True} \mid \mathsf{False} \mid \mathsf{case}\ e\ \mathsf{of}\ \langle \mathsf{True} \to e_1; \mathsf{False} \to e_2 \rangle \\
 & & \mid \mathsf{Nil}_{\tau} \mid \mathsf{Cons}(e_1, e_2) \mid \mathsf{case}\ e\ \mathsf{of}\ \langle \mathsf{Nil} \to e_1; \mathsf{Cons}(x,y) \to e_2 \rangle \\
 & & \mid \mathsf{failure}_{\tau} \mid \mathsf{anything}_{\tau}
\end{array}
$$

Figure 1. Syntax of CuMin

function definitions – each with a single rule $f\ x_1 \ldots x_n = e$ – that can be recursive (even mutually recursive), and as such is a CuMin program. The following function shall exemplify the syntax:

$$
\begin{array}{l}
\mathit{pMap} :: \forall^{\varepsilon}\alpha.\forall^{\varepsilon}\beta.\,(\alpha \to \beta) \to (\alpha,\alpha) \to (\beta,\beta); \\
\mathit{pMap}\ g\ p = \mathsf{case}\ p\ \mathsf{of}\ \langle (u,v) \to (g\ u, g\ v) \rangle
\end{array}
$$

The type system of CuMin contains function types, the naturals, Booleans, lists, and pairs, and is an exemplary subset of the somewhat richer type system of Curry. The missing type features are largely orthogonal to the logic features discussed here and can be treated analogously. Types in CuMin can be polymorphic, but only rank-one polymorphism is allowed (hence, no quantifiers within τs). Curry features Hindley-Milner type inference, while in CuMin polymorphic functions and primitives have to be instantiated explicitly. To this end, type variables are introduced by $\forall$-quantifiers in a function's type annotation and remain in scope throughout its definition.

Figure 2 shows the typing rules for CuMin. An (unordered) typing context $\Gamma = \overline{\alpha_m^{\nu_m}}, \overline{x_n :: \tau_n}$ consists of type variables α with tags $\nu \in \{\varepsilon, *\}$ and term variables with their respective types, which have to be types within Γ. A type is called a *type within* Γ if all its type variables are listed among the $\overline{\alpha_m^{\nu_m}}$. A typing judgment is of the form $\Gamma \vdash e :: \tau$ and states that e is typeable to τ (which must be a type within Γ) under the typing context Γ. If the context in a typing judgment is empty, we simply omit the context. By $[\tau/\alpha]$ we mean syntactic replacement of occurrences of a type variable α by a type τ. We use an analogous notation for terms and we use corresponding vector notations for multiple simultaneous replacements. Figure 3 shows how judgments $\Gamma \vdash \tau \in \mathsf{Data}$ for τ being a data type are derived. Data types are types constructed of base types (Booleans and naturals) and algebraic data type constructors (lists and tuples), but not function types or types containing functions at deeper levels.

Typing of terms is always with respect to a given program P, used in the rule for $f_{\overline{\tau_m}}$ to access type information about a function defined in P. Formally, typing judgments thus would have to be indexed by that program, but we omit the index for the sake of readability. A CuMin program P is well-typed if for each function definition $f\ \overline{x_n} = e$ with type annotation $f :: \forall^{\nu_1}\alpha_1.\cdots\forall^{\nu_m}\alpha_m.\,\tau_1 \to \cdots \to \tau_n \to \tau$ in P the typing judgment $\overline{\alpha_m^{\nu_m}}, \overline{x_n :: \tau_n} \vdash e :: \tau$ holds.

Note, from the rule for $f_{\overline{\tau_m}}$, that the $\forall^{\varepsilon}$ quantifier abstracts over all (monomorphic) types, while the $\forall^{*}$ quantifier only abstracts over data types. As seen from the rule for $\mathsf{anything}_{\tau}$, that primitive (which corresponds to *free variables* in Curry) may be used for data types only. In full Curry the restriction is not imposed by the type system, but implementations may fail at runtime if free variables are used for function types or types containing functions at some deeper level. In this respect CuMin's type system is more static, in order to ease a formal treatment.

To improve readability, we often omit the ε-tags (but never the $*$-tags). We will also take some liberties one would have in Curry, even when actually writing CuMin code: We sometimes omit type instantiations if they can be derived easily, and allow user defined infix operators, which can be translated into proper syntax easily.

$$\Gamma, x :: \tau \vdash x :: \tau \qquad \Gamma \vdash \mathsf{True} :: \mathsf{Bool} \qquad \Gamma \vdash \mathsf{False} :: \mathsf{Bool} \qquad \Gamma \vdash n :: \mathsf{Nat} \qquad \Gamma \vdash \mathsf{Nil}_\tau :: [\tau]$$

$$\frac{\Gamma \vdash e_1 :: \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 :: \tau_1}{\Gamma \vdash e_1\ e_2 :: \tau_2} \qquad \frac{\Gamma \vdash e_1 :: \tau_1 \quad \Gamma, x :: \tau_1 \vdash e_2 :: \tau}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 :: \tau} \qquad \frac{(f :: \forall^{\nu_1} \alpha_1 . \cdots \forall^{\nu_m} \alpha_m . \tau;\ f\ \overline{x_n} = e) \in P}{\Gamma \vdash f_{\overline{\tau_m}} :: \tau[\overline{\tau_m}/\overline{\alpha_m}]}$$ if for all i with $\nu_i = *$ we have $\Gamma \vdash \tau_i \in \mathsf{Data}$

$$\frac{\Gamma \vdash e_1 :: \mathsf{Nat} \quad \Gamma \vdash e_2 :: \mathsf{Nat}}{\Gamma \vdash e_1 + e_2 :: \mathsf{Nat}} \qquad \frac{\Gamma \vdash e_1 :: \mathsf{Nat} \quad \Gamma \vdash e_2 :: \mathsf{Nat}}{\Gamma \vdash e_1 \equiv e_2 :: \mathsf{Bool}} \qquad \frac{\Gamma \vdash e_1 :: \tau_1 \quad \Gamma \vdash e_2 :: \tau_2}{\Gamma \vdash (e_1, e_2) :: (\tau_1, \tau_2)} \qquad \frac{\Gamma \vdash e_1 :: \tau \quad \Gamma \vdash e_2 :: [\tau]}{\Gamma \vdash \mathsf{Cons}(e_1, e_2) :: [\tau]}$$

$$\frac{\Gamma \vdash e :: [\tau'] \quad \Gamma \vdash e_1 :: \tau \quad \Gamma, h :: \tau', t :: [\tau'] \vdash e_2 :: \tau}{\Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ \langle \mathsf{Nil} \to e_1; \mathsf{Cons}(h,t) \to e_2 \rangle :: \tau} \qquad \frac{\Gamma \vdash e :: (\tau_1, \tau_2) \quad \Gamma, l :: \tau_1, r :: \tau_2 \vdash e_1 :: \tau}{\Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ \langle (l,r) \to e_1 \rangle :: \tau}$$

$$\frac{\Gamma \vdash e :: \mathsf{Bool} \quad \Gamma \vdash e_1 :: \tau \quad \Gamma \vdash e_2 :: \tau}{\Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ \langle \mathsf{True} \to e_1; \mathsf{False} \to e_2 \rangle :: \tau} \qquad \Gamma \vdash \mathsf{failure}_\tau :: \tau \qquad \frac{\Gamma \vdash \tau \in \mathsf{Data}}{\Gamma \vdash \mathsf{anything}_\tau :: \tau}$$

Figure 2. Typing rules for CuMin

$$\Gamma, \alpha^* \vdash \alpha \in \mathsf{Data} \qquad \Gamma \vdash \mathsf{Bool} \in \mathsf{Data} \qquad \Gamma \vdash \mathsf{Nat} \in \mathsf{Data}$$

$$\frac{\Gamma \vdash \tau \in \mathsf{Data}}{\Gamma \vdash [\tau] \in \mathsf{Data}} \qquad \frac{\Gamma \vdash \tau \in \mathsf{Data} \qquad \Gamma \vdash \tau' \in \mathsf{Data}}{\Gamma \vdash (\tau, \tau') \in \mathsf{Data}}$$

Figure 3. Rules for being a data type

Let us discuss some specific language features. The term constructor let ... in allows to define local variables and introduces sharing. While these variables may be of function type, they may not be defined by a rule with arguments – i.e., let $f = g$ in ... is fine, but let $f\ x = e$ in ... is not and would have to be given on the top-level instead. Also, and independently of its type, the local variable cannot be used within its own defining expression, which is an actual restriction compared to full Curry.[3]

Logic programming features come into CuMin by certain primitives: Computations that always fail are represented by failure. In contrast to Curry, there are no explicit free variables. Instead, and equivalently, there is the anything-primitive representing every possible value of some type. Since $\mathsf{anything}_\tau$ is an ordinary expression, it can be used as a subexpression at will and all occurrences are evaluated independently. But since let-bindings enforce sharing, in an expression like let $x = \mathsf{anything}_\tau$ in e the variable x can be used multiple times within e to represent the same value at each occurrence. Using the anything-primitive, we can define a binary choice operator $\cup$ that allows to combine two alternatives, like Curry's "?":

$$(\cup) :: \forall^{\epsilon} \alpha . \alpha \to \alpha \to \alpha;$$
$$(\cup)\ x\ y = \mathsf{case}\ \mathsf{anything}_{\mathsf{Bool}}\ \mathsf{of}\ \langle \mathsf{True} \to x; \mathsf{False} \to y \rangle$$

3. CuMin's Type and Term Semantics

In this section we give a denotational semantics for CuMin, based on pointed partially ordered sets (*posets* for short), and discuss the connection to existing semantics for functional-logic languages. Before we do so, we take a look at examples in order to gain some intuition for the kind of nondeterminism we are dealing with.

$coin :: \mathsf{Nat};$	$coin^t :: \{\mathsf{Nat}\};$
$coin = 0 \cup 1$	$coin^t = \{0\} \cup \{1\}$
$double :: \mathsf{Nat} \to \mathsf{Nat};$	$double^t :: \{\mathsf{Nat} \to \{\mathsf{Nat}\}\};$
$double\ n = n + n$	$double^t = \{\lambda n . \{n+n\}\}$
$dc1 :: \mathsf{Nat};$	$dc1^t :: \{\mathsf{Nat}\};$
$dc1 = double\ coin$	$dc1^t = double^t \ni d \bigcup coin^t \ni c \bigcup d\ c$
$dc2 :: \mathsf{Nat};$	$dc2^t :: \{\mathsf{Nat}\};$
$dc2 = coin + coin$	$dc2^t = coin^t \ni c_1 \bigcup coin^t \ni c_2 \bigcup \{c_1 + c_2\}$
$id :: \forall \alpha . \alpha \to \alpha;$	$id^t :: \forall \alpha . \{\alpha \to \{\alpha\}\};$
$id\ x = x$	$id^t = \{\lambda x . \{x\}\}$
$inc :: \mathsf{Nat} \to \mathsf{Nat};$	$inc^t :: \{\mathsf{Nat} \to \{\mathsf{Nat}\}\};$
$inc\ x = x + 1$	$inc^t = \{\lambda x . \{x+1\}\}$
$mayInc1 :: \mathsf{Nat} \to \mathsf{Nat};$	$mayInc1^t :: \{\mathsf{Nat} \to \{\mathsf{Nat}\}\};$
$mayInc1 = id_{\mathsf{Nat}} \cup inc$	$mayInc1^t = id^t_{\mathsf{Nat}} \cup inc^t$
$mayInc2 :: \mathsf{Nat} \to \mathsf{Nat};$	$mayInc2^t :: \{\mathsf{Nat} \to \{\mathsf{Nat}\}\};$
$mayInc2\ x = mayInc1\ x$	$mayInc2^t = \{\lambda x . mayInc1^t \ni m \bigcup m\ x\}$

Figure 4. Some example functions in CuMin (left) and their translations to SaLT (right)

First of all, we have the famous double coin example, given on the upper left of Figure 4. The expression *double coin* will only produce 0 and 2 as results, which exemplifies call-time choice: The two occurrences of the variable n in the body of *double* represent one (shared) value that *coin* can evaluate to, rather than the expression *coin* itself. In contrast, $coin + coin$ does allow 1 as a result, because the top-level symbol *coin* is evaluated twice and every such evaluation is independent. Another way (beside passing an argument to a function) to ensure sharing is using let-bindings as in let $c = coin$ in $c + c$. Here c also is a variable and thus yields the same value twice. Note that replacing both occurrences of c by the defining expression *coin* would change the outcome (see above), because it breaks sharing (unlike in Haskell, where even top-level constants are shared). To model call-time choice in our denotational semantics, variables will have single values assigned to them despite the fact that expressions will have a set-valued semantics.

However, call-time choice does not imply that the argument needs to produce any result in order for the function to produce at least one. If we define a function

$$alwaysTrue :: \mathsf{Bool} \to \mathsf{Bool};$$
$$alwaysTrue\ x = \mathsf{True}$$

[3] For example, in Curry the definition let $ones = [\,]\,?(1 : ones)$ in *ones* will only produce two lists: the empty list and an infinite sequence of 1s. This is due to the fact that as soon as the second alternative has been chosen when evaluating to head normal form, the variable *ones* refers to the selfsame object in the recursive call, is thus already determined and cannot switch back to being the empty list. While in an operational semantics this behavior is quite natural, it hinders a compositional denotational semantics. Thus we do not allow this in CuMin (see the typing rule for let ... in in Figure 2, preventing the newly defined variable from being used within its own defining expression). Also, note that while [13] showed the equivalence for two established semantics, one operational [1, 2] and one based on rewriting logic [9], that semantic investigation was also performed in the absence of recursive let-bindings only.

$$[\![e_1 + e_2]\!]^i_{\theta,\sigma} = \{\mathbf{a} +^{\perp} \mathbf{b} \mid \mathbf{a} \in [\![e_1]\!]^i_{\theta,\sigma}, \mathbf{b} \in [\![e_2]\!]^i_{\theta,\sigma}\} \qquad [\![(e_1,e_2)]\!]^i_{\theta,\sigma} = \{\perp\} \cup ([\![e_1]\!]^i_{\theta,\sigma} \times [\![e_2]\!]^i_{\theta,\sigma})$$

$$[\![e_1 \equiv e_2]\!]^i_{\theta,\sigma} = \{\mathbf{a} =^{\perp} \mathbf{b} \mid \mathbf{a} \in [\![e_1]\!]^i_{\theta,\sigma}, \mathbf{b} \in [\![e_2]\!]^i_{\theta,\sigma}\} \qquad [\![e_1\ e_2]\!]^i_{\theta,\sigma} = \bigcup_{\mathbf{f} \in [\![e_1]\!]^i_{\theta,\sigma}} \bigcup_{\mathbf{a} \in [\![e_2]\!]^i_{\theta,\sigma}} \mathbf{f}\,\mathbf{a}$$

$$[\![\mathsf{Cons}(e_1,e_2)]\!]^i_{\theta,\sigma} = \{\perp\} \cup \{\mathbf{h} : \mathbf{t} \mid \mathbf{h} \in [\![e_1]\!]^i_{\theta,\sigma}, \mathbf{t} \in [\![e_2]\!]^i_{\theta,\sigma}\} \qquad [\![\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2]\!]^i_{\theta,\sigma} = \bigcup_{\mathbf{x} \in [\![e_1]\!]^i_{\theta,\sigma}} [\![e_2]\!]^i_{\theta,\sigma[x \mapsto \mathbf{x}]}$$

$$[\![x]\!]^i_{\theta,\sigma} = \downarrow\sigma(x)$$

$$[\![n]\!]^i_{\theta,\sigma} = \{\perp, \mathbf{n}\} \qquad [\![\mathsf{case}\ e\ \mathsf{of}\ \langle \mathsf{True} \to e_1; \mathsf{False} \to e_2\rangle]\!]^i_{\theta,\sigma} = \bigcup_{\mathbf{b} \in [\![e]\!]^i_{\theta,\sigma}} \begin{cases} \{\perp\} & \text{if } \mathbf{b} = \perp \\ [\![e_1]\!]^i_{\theta,\sigma} & \text{if } \mathbf{b} = \mathbf{True} \\ [\![e_2]\!]^i_{\theta,\sigma} & \text{if } \mathbf{b} = \mathbf{False} \end{cases}$$

$$[\![\mathsf{True}]\!]^i_{\theta,\sigma} = \{\perp, \mathbf{True}\}$$

$$[\![\mathsf{False}]\!]^i_{\theta,\sigma} = \{\perp, \mathbf{False}\}$$

$$[\![\mathsf{Nil}_\tau]\!]^i_{\theta,\sigma} = \{\perp, []\} \qquad [\![\mathsf{case}\ e\ \mathsf{of}\ \langle \mathsf{Nil} \to e_1; \mathsf{Cons}(h,t) \to e_2\rangle]\!]^i_{\theta,\sigma} = \bigcup_{\mathbf{l} \in [\![e]\!]^i_{\theta,\sigma}} \begin{cases} \{\perp\} & \text{if } \mathbf{l} = \perp \\ [\![e_1]\!]^i_{\theta,\sigma} & \text{if } \mathbf{l} = [] \\ [\![e_2]\!]^i_{\theta,\sigma[h \mapsto \mathbf{h}, t \mapsto \mathbf{t}]} & \text{if } \mathbf{l} = \mathbf{h} : \mathbf{t} \end{cases}$$

$$[\![\mathsf{failure}_\tau]\!]^i_{\theta,\sigma} = \{\perp\}$$

$$[\![\mathsf{anything}_\tau]\!]^i_{\theta,\sigma} = [\![\tau]\!]_\theta \qquad [\![\mathsf{case}\ e\ \mathsf{of}\ \langle (l,r) \to e_1\rangle]\!]^i_{\theta,\sigma} = \bigcup_{\mathbf{p} \in [\![e]\!]^i_{\theta,\sigma}} \begin{cases} \{\perp\} & \text{if } \mathbf{p} = \perp \\ [\![e_1]\!]^i_{\theta,\sigma[l \mapsto \mathbf{l}, r \mapsto \mathbf{r}]} & \text{if } \mathbf{p} = (\mathbf{l},\mathbf{r}) \end{cases}$$

$$[\![f_{\overline{\tau_m}}]\!]^0_{\theta,\sigma} = \{\perp\}$$

$$[\![f_{\overline{\tau_m}}]\!]^{i+1}_{\theta,\sigma} = \downarrow(\lambda \mathbf{a}_1. \cdots \downarrow(\lambda \mathbf{a}_n. [\![e]\!]^i_{\overline{[\alpha_m \mapsto [\![\tau_m]\!]_\theta]}, \overline{[x_n \mapsto \mathbf{a}_n]}}) \cdots) \quad \text{with } f :: \forall^{\nu_1} \alpha_1. \cdots \forall^{\nu_m} \alpha_m. \tau;\ f\ \overline{x_n} = e\ \text{in}\ P$$

Figure 5. Denotational term semantics for CuMin

and then apply it as in *alwaysTrue* failure, the result will still be True. The same is the case for variables that are introduced by let-bindings or patterns in case expressions. For example, the expression case (failure, failure) of $\langle (x,y) \mapsto \mathsf{True}\rangle$ does allow True as a result, since neither x nor y is actually used. There are only two kinds of constructs that are strict (i.e., fail if the argument does): Case expressions are strict in the scrutinee and primitive operations (like addition or the equality test) are strict in all arguments. Therefore, case failure of $\langle (x,y) \mapsto \mathsf{True}\rangle$ does not produce a result.

In the denotational semantics, failure is represented already on the element level, in order to allow variables to hold partial values. These partial values are ordered by a definedness relation, which makes the semantic ranges partially ordered sets.

We also have to take care of recursion. In Haskell the capability of unrestricted recursion allows to define infinite lists, which would not appear in a similar language with only structural recursion (corresponding to *fold*s). Even though expressions defining infinite lists can never be fully evaluated, they have to have some semantic value. This usually requires switching from a semantics based on sets to one based on some kind of domain, i.e., ordered sets with a limit structure – e.g., directed complete partial orders, or *dcpos* for short. So one way to combine recursion and nondeterminism is to build a set level on top of a dcpo-structure by using power domains as done in [5]. However, this makes the technicalities rather intricate. The key insight to evade these complications here is that nondeterminism actually subsumes general fixpoints. Indeed, unrestricted recursion can be recovered from just a form of structural recursion, failure, and free variables. For example,

$$\begin{array}{l} \mathit{replicates} :: \forall\alpha.\, \alpha \to [\alpha]; \\ \mathit{replicates}\ x = \mathsf{Nil}_\alpha \cup \mathsf{Cons}(x, \mathit{replicates}_\alpha\ x) \end{array}$$

can also be defined by

$$\begin{array}{l} \mathit{gen} :: \forall\alpha.\, (\alpha \to [\alpha]) \to \alpha \to [\alpha]; \\ \mathit{gen}\ r\ x = \mathsf{Nil}_\alpha \cup \mathsf{Cons}(x, r\ x); \\ \mathit{replicates} :: \forall\alpha.\, \alpha \to [\alpha]; \\ \mathit{replicates}\ x = \mathsf{let}\ n = \mathsf{anything}_{\mathsf{Nat}}\ \mathsf{in} \\ \qquad\qquad \mathsf{foldn}\ \mathit{gen}_\alpha\ \mathsf{failure}_{\alpha \to [\alpha]}\ n\ x \end{array}$$

where foldn $h\ x\ m$ is m-fold application of h to x, and thus structural recursion over the naturals.[4]

[4] The general idea here is to replace $f\ x = e$, where e contains f, by $\mathit{gen}\ r\ x = e'$, where e' is e with f replaced by r, and $f\ x = \mathsf{let}\ n = \mathsf{anything}_{\mathsf{Nat}}\ \mathsf{in}\ \mathsf{foldn}\ \mathit{gen}\ \mathsf{failure}\ n\ x$.

We do not actually eschew general recursion in favor of only a structural recursion primitive in the syntax, but the just given explanations indicate that no extra provision is necessary for fixpoints in the semantics. Indeed, it is enough for the element level to allow failure, while the set level takes care of nondeterminism and thus automatically of limits. It suffices to draw elements from pointed posets rather than from dcpos since the fixpoint construction will not take place on the element level anyway, but on the set level, and our power set construction will automatically turn a pointed poset at the element level into a dcpo (indeed a semilattice) at the set level, so there is actually no need to require a dcpo structure already at the element level, as [5, 6] did.

So, for the element level in our semantics here, types are interpreted as pointed posets $(P, \sqsubseteq)$, whose least element will always be denoted by $\perp$. Since all posets we are dealing with are pointed, we simply write *poset* rather than pointed poset from now on. Order-preserving functions are called *monotone* and this is not meant to imply being a pointed function as well.

Then, we call a subset A of a poset P a *lower set* if $\perp \in A$ and for all $\mathbf{x} \in A$ and $\mathbf{y} \in P$ we have that $\mathbf{y} \sqsubseteq \mathbf{x}$ implies $\mathbf{y} \in A$. If $\mathbf{x}$ is an element of P, we write $\downarrow\mathbf{x} = \{\mathbf{y} \in P \mid \mathbf{y} \sqsubseteq \mathbf{x}\}$ for the lower set of elements smaller than or equal to $\mathbf{x}$. For the power set construction, the set of all lower sets of a poset P is denoted by $\mathscr{P}_\ell(P)$. It is itself partially ordered by set inclusion $\subseteq$, making $\{\perp\}$ the least element $\perp$ (since $\emptyset$ is no lower set). The function sending $\mathbf{x} \in P$ to $\downarrow\mathbf{x} \in \mathscr{P}_\ell(P)$ is monotone under this order (and also pointed). While P itself does not have to have any suprema, $\mathscr{P}_\ell(P)$ contains suprema of arbitrary collections (their union), because a union of lower sets will again be a lower set. If the collection happens to be empty, its union is defined to be the set $\{\perp\}$. Thus $\mathscr{P}_\ell(P)$ is a join semilattice and thus also a dcpo.

To provide for polymorphic types, the type semantics is defined w.r.t. a type environment θ assigning posets to type variables. The defining equations are:

$$[\![\mathsf{Bool}]\!]_\theta = \{\mathbf{True}, \mathbf{False}\}_\perp \qquad [\![\mathsf{Nat}]\!]_\theta = \mathbb{N}_\perp \qquad [\![\alpha]\!]_\theta = \theta(\alpha)$$

$$[\![[\tau]]\!]_\theta = \{\mathbf{x}_1 : \cdots : \mathbf{x}_n : \mathbf{e} \mid n \geq 0, \mathbf{x}_i \in [\![\tau]\!]_\theta, \mathbf{e} \in \{\perp, []\}\}$$

$$[\![(\tau,\tau')]\!]_\theta = \{(\mathbf{l},\mathbf{r}) \mid \mathbf{l} \in [\![\tau]\!]_\theta, \mathbf{r} \in [\![\tau']\!]_\theta\}_\perp$$

$$[\![\tau \to \tau']\!]_\theta = \{\mathbf{f} : [\![\tau]\!]_\theta \to \mathscr{P}_\ell([\![\tau']\!]_\theta) \mid \mathbf{f}\ \text{monotone}\}$$

The sets $\{\mathbf{True}, \mathbf{False}\}$ and $\mathbb{N}$ are discretely ordered and the lifting operation $\cdot_\perp$ adds $\perp$ as least element. Functions map *one element* of the input type to *a set of elements* of the output type. The order on the

function space is pointwise and the least defined function $\lambda\mathbf{a}.\{\bot\}$ serves as least element $\bot$. In the semantic interpretation of lists and tuples, entries are single elements, not sets. If $\mathbf{x} = \mathbf{x}_1 : \cdots : \mathbf{x}_n : \mathbf{e}_\mathbf{x}$ and $\mathbf{y} = \mathbf{y}_1 : \cdots : \mathbf{y}_m : \mathbf{e}_\mathbf{y}$ with $\mathbf{e}_\mathbf{x}, \mathbf{e}_\mathbf{y} \in \{\bot, []\}$, then $\mathbf{x} \sqsubseteq \mathbf{y}$ holds if $n = m$ and $\mathbf{e}_\mathbf{y} = []$ and $\mathbf{x}_i \sqsubseteq \mathbf{y}_i$ for all $1 \leq i \leq n$, or $n \leq m$ and $\mathbf{e}_\mathbf{x} = \bot$ and (again) $\mathbf{x}_i \sqsubseteq \mathbf{y}_i$ for all $1 \leq i \leq n$. The order on tuples is entrywise and the type interpretation contains an explicit $\bot$ that is not a tuple.

The term semantics is defined in two stages, first $[\![e]\!]^i_{\theta,\sigma}$ in Figure 5, with step index i (and w.r.t. a fixed program P). The step index restricts the number of nested function calls that can be performed. Every further invocation of a function symbol will simply result in failure. Only well-typed (in some typing context Γ) terms are considered. As before, θ is some type environment assigning posets to type variables. We also need a term environment σ mapping (typed) term variables $x :: \tau$ in Γ to elements $\sigma(x) \in [\![\tau]\!]_\theta$. Environments are written as a vector of assignments $[\overline{x_i \mapsto \mathbf{v}_i}]$ or can be given in terms of an already defined environment extended by some new bindings: $\sigma[x \mapsto \mathbf{v}]$. Empty environments are written as $\emptyset$. For the semantics of addition and of the equality test, we employ the strict extension of the usual addition and equality test on naturals, in particular evaluating to failure if one of the arguments does. If $\Gamma \vdash e :: \tau$ holds, then $[\![e]\!]^i_{\theta,\sigma}$ is a lower subset of $[\![\tau]\!]_\theta$ for every $i \in \mathbb{N}$. The semantics is monotone w.r.t. the variable bindings and the index i. This means that more defined values $\sigma(x)$ as well as increasing the index i can only lead to more possible results.

The second stage is to define the semantics (without restriction on the number of nested function calls) of an expression as:

$$[\![e]\!]_{\theta,\sigma} = \bigcup_{i \in \mathbb{N}} [\![e]\!]^i_{\theta,\sigma} \tag{1}$$

This is still monotone w.r.t. the variable bindings.

The semantics presented here is a conceptual simplification, simpler but equally descriptive and expressive, of the denotational semantics based on dcpos given (for a very similar language, called TFLC) by [5, 6]. For the language FlatCurry used in implementations we mentioned at the beginning of Section 2, there are an established operational semantics [1, 2] and an established rewriting logic semantics [9]. Note that [13] showed, without considering recursive let-bindings, that the operational semantics and the rewriting logic semantics are equivalent. So we would be in good company with our semantics for that same setting if we were to formally connect it to either of those latter two semantics. Indeed, the denotational semantics given here can be easily adapted to FlatCurry without recursive let-bindings (only minor syntactic differences remain) and we claim the resulting semantics to be adequate with respect to the operational and rewriting logic semantics. The formal proof of this claim (specifically using the semantics given in [2] as reference point) is current work.

We have already discussed, at the beginning of this section, that replacing variables by their defining expressions can change the semantics. The same is true for inlining of function definitions, as could be seen by comparing the semantics of the two CuMin-functions *dc1* and *dc2* (the latter of which is the former with *double* inlined) given in Figure 4. The figure also features an example concerning eta-equivalence: The two CuMin-functions *mayInc1* and *mayInc2* are not semantically equivalent, which disproves validity of eta-equivalence as a law (for CuMin, as well as for Curry). Note that the difference between the two functions can only be observed when using either as an argument for a higher-order function like *pMap*. While *pMap mayInc1* $(0,0)$ can only produce $(0,0)$ and $(1,1)$ as results, *pMap mayInc2* $(0,0)$ can also result in $(0,1)$ and $(1,0)$. Such intricacies make equational reasoning nearly impossible in CuMin. The situation is nicer in SaLT, introduced next.

$$\begin{array}{rcl}
P & ::= & D; P \mid D \\
D & ::= & f :: \kappa\,\tau;\ f = e \\
\kappa & ::= & \forall^e \alpha.\kappa \mid \forall^* \alpha.\kappa \mid \varepsilon \\
\tau & ::= & \alpha \mid \mathsf{Bool} \mid \mathsf{Nat} \mid [\tau] \mid (\tau, \tau') \mid \tau \to \tau' \mid \{\tau\} \\
e & ::= & x \mid \lambda\, x :: \tau.e \mid f_{\overline{\tau_m}} \mid e_1\ e_2 \mid n \mid e_1 + e_2 \mid e_1 \equiv e_2 \\
 & & \mid (e_1, e_2) \mid \mathsf{case}\ e\ \mathsf{of}\ \langle (x,y) \to e_1 \rangle \\
 & & \mid \mathsf{True} \mid \mathsf{False} \mid \mathsf{case}\ e\ \mathsf{of}\ \langle \mathsf{True} \to e_1; \mathsf{False} \to e_2 \rangle \\
 & & \mid \mathsf{Nil}_\tau \mid \mathsf{Cons}(e_1, e_2) \mid \mathsf{case}\ e\ \mathsf{of}\ \langle \mathsf{Nil} \to e_1; \mathsf{Cons}(x,y) \to e_2 \rangle \\
 & & \mid \{e\} \mid e_1 \ni x \bigcup e_2 \mid \mathsf{failure}_\tau \mid \mathsf{anything}_\tau
\end{array}$$

Figure 6. Syntax of SaLT

$$\frac{\Gamma, x :: \tau_1 \vdash e :: \tau_2}{\Gamma \vdash \lambda\, x :: \tau_1.e :: \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash \tau \in \mathsf{Data}}{\Gamma \vdash \mathsf{anything}_\tau :: \{\tau\}}$$

$$\frac{\Gamma \vdash e_1 :: \tau}{\Gamma \vdash \{e_1\} :: \{\tau\}} \qquad \frac{\Gamma \vdash e_1 :: \{\tau_1\} \qquad \Gamma, x :: \tau_1 \vdash e_2 :: \{\tau\}}{\Gamma \vdash e_1 \ni x \bigcup e_2 :: \{\tau\}}$$

Figure 7. Additional/replacement typing rules for SaLT

4. The Language SaLT and its Semantics

The syntax for SaLT is given in Figure 6. It is largely the same as that of CuMin, the major difference being that nondeterminism is made explicit by using sets on the syntactic level. For example, the following function applies f to every element x of the set s.

$$\begin{array}{l}
sMap :: \forall\alpha.\forall\beta.(\alpha \to \beta) \to \{\alpha\} \to \{\beta\}; \\
sMap = \lambda\, f :: \alpha \to \beta.\lambda\, s :: \{\alpha\}.s \ni x \bigcup \{f\ x\}
\end{array}$$

As in CuMin, top-level SaLT functions can be (mutually) recursive.

The typing rules for SaLT are mostly the same as for CuMin (see Figure 2), subject to some adjustments: We have added a set type constructor $\{\tau\}$, which is a lot like the IO type constructor in Haskell or Curry in the sense that there is no way to inspect sets[5]; they can only be combined using language primitives, like the newly introduced singleton sets $\{e\}$ and indexed unions $e_1 \ni x \bigcup e_2$ (their typing rules are in Figure 7). The latter introduces a new variable x which is in scope throughout e_2, just like the mathematical notation $\bigcup_{x \in e_1} e_2$ does.[6] The primitive anything that CuMin provided for nondeterminism is now interpreted as acting at set types (see Figure 7). There is no let ... in, and top-level functions cannot list formal parameters in SaLT (see Figure 6), so these constructs have to be paraphrased using lambda-abstractions. There is no inherent reason against keeping either as in CuMin, but they do not occur in our translation procedure and are therefore unnecessary. Accordingly, the let ... in typing rule from Figure 2 is deleted, the rule given there for function symbols is restricted to the case $n = 0$ for SaLT, and the added syntactic form of lambda-abstraction has the usual typing rule (see Figure 7). The membership rules for being a data type (cf. Figure 3) remain unchanged. In particular, set types are not considered to live in Data, so that anything cannot be used to produce nested sets.

A SaLT program P is well-typed if for each function definition $f = e$ with type annotation $f :: \forall^{\nu_1}\alpha_1.\cdots\forall^{\nu_m}\alpha_m.\tau$ in P the typing judgment $\overline{\alpha_m^{\nu_m}} \vdash e :: \tau$ holds.

When writing code in SaLT, we take the same liberties as for CuMin, i.e., omit type annotations and allow user defined infix operators. We sometimes use set brackets as a unary operator,

[5] Sets are a monad that cannot be escaped.

[6] The expression e_1 is written on the left in our notation in order to avoid nested indices and parentheses.

$$
\begin{aligned}
[\![e_1 + e_2]\!]^i_{\theta,\sigma} &= [\![e_1]\!]^i_{\theta,\sigma} +^{\perp} [\![e_2]\!]^i_{\theta,\sigma} \\
[\![e_1 \equiv e_2]\!]^i_{\theta,\sigma} &= [\![e_1]\!]^i_{\theta,\sigma} =^{\perp} [\![e_2]\!]^i_{\theta,\sigma} \\
[\![\mathsf{Cons}(e_1,e_2)]\!]^i_{\theta,\sigma} &= [\![e_1]\!]^i_{\theta,\sigma} : [\![e_2]\!]^i_{\theta,\sigma} \\
[\![(e_1,e_2)]\!]^i_{\theta,\sigma} &= ([\![e_1]\!]^i_{\theta,\sigma}, [\![e_2]\!]^i_{\theta,\sigma}) \\
[\![e_1\ e_2]\!]^i_{\theta,\sigma} &= [\![e_1]\!]^i_{\theta,\sigma}\ [\![e_2]\!]^i_{\theta,\sigma} \\
[\![\{e\}]\!]^i_{\theta,\sigma} &= \downarrow [\![e]\!]^i_{\theta,\sigma} \\
[\![x]\!]^i_{\theta,\sigma} &= \sigma(x) \\
[\![n]\!]^i_{\theta,\sigma} &= \mathbf{n} \\
[\![\mathsf{True}]\!]^i_{\theta,\sigma} &= \mathbf{True} \\
[\![\mathsf{False}]\!]^i_{\theta,\sigma} &= \mathbf{False} \\
[\![\mathsf{Nil}_\tau]\!]^i_{\theta,\sigma} &= [\,] \\
[\![\mathsf{failure}_\tau]\!]^i_{\theta,\sigma} &= \perp \\
[\![\mathsf{anything}_\tau]\!]^i_{\theta,\sigma} &= [\![\tau]\!]_\theta
\end{aligned}
\qquad
\begin{aligned}
[\![e_1 \ni x \bigcup e_2]\!]^i_{\theta,\sigma} &= \textstyle\bigcup_{\mathbf{x} \in [\![e_1]\!]^i_{\theta,\sigma}} [\![e_2]\!]^i_{\theta,\sigma[x \mapsto \mathbf{x}]} \\
[\![\lambda x :: \tau.\, e]\!]^i_{\theta,\sigma} &= \lambda \mathbf{x}. [\![e]\!]^i_{\theta,\sigma[x \mapsto \mathbf{x}]} \\
[\![\mathsf{case}\ e\ \mathsf{of}\ \langle \mathsf{True} \to e_1; \mathsf{False} \to e_2 \rangle]\!]^i_{\theta,\sigma} &= \begin{cases} \perp & \text{if } [\![e]\!]^i_{\theta,\sigma} = \perp \\ [\![e_1]\!]^i_{\theta,\sigma} & \text{if } [\![e]\!]^i_{\theta,\sigma} = \mathbf{True} \\ [\![e_2]\!]^i_{\theta,\sigma} & \text{if } [\![e]\!]^i_{\theta,\sigma} = \mathbf{False} \end{cases} \\
[\![\mathsf{case}\ e\ \mathsf{of}\ \langle \mathsf{Nil} \to e_1; \mathsf{Cons}(h,t) \to e_2 \rangle]\!]^i_{\theta,\sigma} &= \begin{cases} \perp & \text{if } [\![e]\!]^i_{\theta,\sigma} = \perp \\ [\![e_1]\!]^i_{\theta,\sigma} & \text{if } [\![e]\!]^i_{\theta,\sigma} = [\,] \\ [\![e_2]\!]^i_{\theta,\sigma[h \mapsto \mathbf{h}, t \mapsto \mathbf{t}]} & \text{if } [\![e]\!]^i_{\theta,\sigma} = \mathbf{h} : \mathbf{t} \end{cases} \\
[\![\mathsf{case}\ e\ \mathsf{of}\ \langle (l,r) \to e_1 \rangle]\!]^i_{\theta,\sigma} &= \begin{cases} \perp & \text{if } [\![e]\!]^i_{\theta,\sigma} = \perp \\ [\![e_1]\!]^i_{\theta,\sigma[l \mapsto \mathbf{l}, r \mapsto \mathbf{r}]} & \text{if } [\![e]\!]^i_{\theta,\sigma} = (\mathbf{l},\mathbf{r}) \end{cases} \\
[\![f_{\overline{\tau_m}}]\!]^0_{\theta,\sigma} &= \perp \\
[\![f_{\overline{\tau_m}}]\!]^{i+1}_{\theta,\sigma} &= [\![e]\!]^i_{\overline{[\alpha_m \mapsto [\![\tau_m]\!]_\theta]},\emptyset} \qquad \text{with } f :: \forall^{\nu_1} \alpha_1 . \cdots \forall^{\nu_m} \alpha_m .\, \tau;\ f = e \text{ in } P
\end{aligned}
$$

Figure 8. Denotational term semantics for SaLT

employing a polymorphic top-level function

$$
\begin{aligned}
&\{\_\} :: \forall \alpha.\, \alpha \to \{\alpha\}; \\
&\{\_\} = \lambda x :: \alpha.\, \{x\}
\end{aligned}
$$

We also define a binary choice operator as in CuMin:

$$
\begin{aligned}
&(\cup) :: \forall \alpha.\, \{\alpha\} \to \{\alpha\} \to \{\alpha\}; \\
&(\cup) = \lambda x.\, \lambda y.\, \mathsf{anything}_{\mathsf{Bool}} \ni b \bigcup \mathsf{case}\ b\ \mathsf{of}\ \langle \mathsf{True} \to x; \mathsf{False} \to y \rangle
\end{aligned}
$$

There are two changes in the semantics of types:

$$
\begin{aligned}
[\![\tau \to \tau']\!]_\theta &= \{\mathbf{f} : [\![\tau]\!]_\theta \to [\![\tau']\!]_\theta \mid \mathbf{f} \text{ monotone}\} \\
[\![\{\tau\}]\!]_\theta &= \mathscr{P}_\ell([\![\tau]\!]_\theta)
\end{aligned}
$$

The first definition changes the interpretation of the function type, which now maps values in $[\![\tau]\!]_\theta$ to single values in $[\![\tau']\!]_\theta$ instead of to lower sets. The lower set construction only appears in the interpretation of the set type, in the second and newly introduced definition. The other defining equations of the type semantics are taken over from CuMin, but replacing semantics brackets $[\![\cdot]\!]$ by $[\![\cdot]\!]$.

These brackets are also used for the term semantics of SaLT as given in Figure 8, where the term environment σ maps variables to single values (just like in CuMin). If $\Gamma \vdash e :: \tau$ holds, then $[\![e]\!]^i_{\theta,\sigma}$ is an element of $[\![\tau]\!]_\theta$ for every $i \in \mathbb{N}$ (unlike in CuMin, where it would be a subset). As for CuMin, the semantics is monotone w.r.t. σ and i. This means that binding variables to more defined values or increasing the index will lead to at least as defined results.

Only for set-typed expressions we define the limit:

$$
[\![e]\!]_{\theta,\sigma} = \bigcup_{i \in \mathbb{N}} [\![e]\!]^i_{\theta,\sigma} \tag{2}
$$

As for CuMin, this is still monotone w.r.t. the variable bindings.

We define two terms $\Gamma \vdash t_1 :: \tau$ and $\Gamma \vdash t_2 :: \tau$ to be *semantically equivalent*, written as $t_1 \equiv t_2$, if they are interchangeable as subexpressions (in the sense of preserving the semantics w.r.t. arbitrary environments) of arbitrary set-typed expressions. If the terms are themselves set-typed, this is equivalent to $[\![t_1]\!]_{\theta,\sigma} = [\![t_2]\!]_{\theta,\sigma}$ for all environments θ, σ – because of the compositionality of the semantics. Otherwise, $[\![\{t_1\}]\!]_{\theta,\sigma} = [\![\{t_2\}]\!]_{\theta,\sigma}$ for all environments θ, σ is obviously necessary and also sufficient because of compositionality and equation (3) below.

In contrast to what happens in CuMin, in SaLT common manipulations of expressions such as beta- and eta-reduction or inlining of definitions lead to semantically equivalent terms. Beta- and eta-reduction do not at all change what the semantics functions $[\![\cdot]\!]^i$ and $[\![\cdot]\!]$ compute, because syntactic function applications in SaLT represent actual function application, which is not true for CuMin or full Curry. Inlining of a function definition $f = e$ may change what the semantics functions compute, since $[\![f_{\overline{\tau_m}}]\!]^0_{\theta,\sigma}$ is $\perp$ by definition, while $[\![e[\overline{\tau_m/\alpha_m}]]\!]^0_{\theta,\sigma}$ can be a proper value. But the two terms *are* semantically equivalent according to the general notion:

$$
\begin{aligned}
[\![\{f_{\overline{\tau_m}}\}]\!]_{\theta,\sigma} &= \bigcup_{i \in \mathbb{N}} \downarrow([\![f_{\overline{\tau_m}}]\!]^i_{\theta,\sigma}) \\
&= \downarrow\perp \cup \bigcup_{i \in \mathbb{N}} \downarrow([\![e]\!]^i_{\overline{[\alpha_m \mapsto [\![\tau_m]\!]_\theta]},\emptyset}) \\
&= [\![\{e[\overline{\tau_m/\alpha_m}]\}]\!]_{\theta,\sigma}
\end{aligned}
$$

Moreover, there are interesting equivalences involving the additional concepts: the set type and the primitives using it. Using the above notion of semantic equivalence, we can state the three monad laws for sets:

$$
\{e_1\} \ni x \bigcup e_2 \ \equiv\ e_2[e_1/x] \tag{3}
$$

$$
e_1 \ni x \bigcup \{x\} \ \equiv\ e_1 \tag{4}
$$

$$
(e_1 \ni x \bigcup e_2) \ni y \bigcup e_3 \ \equiv\ e_1 \ni x \bigcup (e_2 \ni y \bigcup e_3) \tag{5}
$$

The first one will be used frequently when simplifying translated code. The third one shows the parentheses to be superfluous and we will indeed omit them. Also, if neither x appears in e_2, nor y in e_1, indexed unions can be swapped:

$$
e_1 \ni x \bigcup e_2 \ni y \bigcup e_3 \ \equiv\ e_2 \ni y \bigcup e_1 \ni x \bigcup e_3 \tag{6}
$$

The equations (3)–(6) are proved by unfolding (2) and the definitions in Figure 8 and then using the corresponding properties on the semantic level.

Also, it is not difficult to see that the binary choice operator $\cup$ is associative and $\mathsf{failure}_{\{\tau\}} \equiv \{\mathsf{failure}_\tau\}$ (for the relevant τ) is its unit. We also have a distributivity and an idempotence law:

$$
(e_1 \cup e_2) \ni x \bigcup e_3 \ \equiv\ (e_1 \ni x \bigcup e_3) \cup (e_2 \ni x \bigcup e_3) \tag{7}
$$

$$
e \cup e \ \equiv\ e \tag{8}
$$

The set type constructor is not an additive monad, though, since in general $\{\mathsf{failure}\} \ni x \bigcup e \not\equiv \{\mathsf{failure}\}$. By the first monad law, the left-hand side here is equivalent to $e[\mathsf{failure}/x]$, and if e does not make use of the variable x, that will not in general be equivalent to $\{\mathsf{failure}\}$. In this respect our monad is rather like the one in [3] than

$$
\begin{array}{rcl}
\lceil x \rceil &=& \{x\} \\
\lceil n \rceil &=& \{n\} \\
\lceil \mathsf{True} \rceil &=& \{\mathsf{True}\} \\
\lceil \mathsf{False} \rceil &=& \{\mathsf{False}\} \\
\lceil \mathsf{failure}_\tau \rceil &=& \{\mathsf{failure}_{\lfloor\tau\rfloor}\} \\
\lceil \mathsf{anything}_\tau \rceil &=& \mathsf{anything}_{\lfloor\tau\rfloor}
\end{array}
\qquad
\begin{array}{rcl}
\lceil \mathsf{Nil}_\tau \rceil &=& \{\mathsf{Nil}_{\lfloor\tau\rfloor}\} \\
\lceil f_{\overline{\tau_m}} \rceil &=& f^t_{\overline{\lfloor\tau_m\rfloor}} \\
\lceil \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \rceil &=& \lceil e_1 \rceil \ni x \bigcup \lceil e_2 \rceil \\
\lceil e_1\ e_2 \rceil &=& \lceil e_1 \rceil \ni x_1 \bigcup \lceil e_2 \rceil \ni x_2 \bigcup x_1\ x_2
\end{array}
$$

$$
\begin{array}{rcl}
\lceil \mathsf{Cons}(e_1,e_2) \rceil &=& \lceil e_1 \rceil \ni x_1 \bigcup \lceil e_2 \rceil \ni x_2 \bigcup \{\mathsf{Cons}(x_1,x_2)\} \\
\lceil e_1 + e_2 \rceil &=& \lceil e_1 \rceil \ni x_1 \bigcup \lceil e_2 \rceil \ni x_2 \bigcup \{x_1 + x_2\} \\
\lceil e_1 \equiv e_2 \rceil &=& \lceil e_1 \rceil \ni x_1 \bigcup \lceil e_2 \rceil \ni x_2 \bigcup \{x_1 \equiv x_2\} \\
\lceil (e_1,e_2) \rceil &=& \lceil e_1 \rceil \ni x_1 \bigcup \lceil e_2 \rceil \ni x_2 \bigcup \{(x_1,x_2)\}
\end{array}
$$

$$
\begin{array}{rcl}
\lceil \mathsf{case}\ e\ \mathsf{of}\ \langle \mathsf{True} \to e_1; \mathsf{False} \to e_2 \rangle \rceil &=& \lceil e \rceil \ni b \bigcup \mathsf{case}\ b\ \mathsf{of}\ \langle \mathsf{True} \to \lceil e_1 \rceil; \mathsf{False} \to \lceil e_2 \rceil \rangle \\
\lceil \mathsf{case}\ e\ \mathsf{of}\ \langle \mathsf{Nil} \to e_1; \mathsf{Cons}(h,t) \to e_2 \rangle \rceil &=& \lceil e \rceil \ni l \bigcup \mathsf{case}\ l\ \mathsf{of}\ \langle \mathsf{Nil} \to \lceil e_1 \rceil; \mathsf{Cons}(h,t) \to \lceil e_2 \rceil \rangle \\
\lceil \mathsf{case}\ e\ \mathsf{of}\ \langle (l,r) \to e_1 \rangle \rceil &=& \lceil e \rceil \ni p \bigcup \mathsf{case}\ p\ \mathsf{of}\ \langle (l,r) \to \lceil e_1 \rceil \rangle
\end{array}
$$

Figure 9. Translation from CuMin expressions to SaLT expressions

(for example) the plain list monad. However, our translation, which is given in the next section, is different from the one given in [3] (which also has a different aim).

5. Translating CuMin into SaLT

We give a purely syntactic way of translating CuMin programs into SaLT programs having the same semantics. Programs are translated function by function. Specifically,

$$f :: \kappa\ \tau_1 \to \cdots \to \tau_n \to \tau;\quad f\ \overline{x_n} = e$$

as a CuMin function definition is translated into SaLT as

$$f^t :: \kappa\ \{\lfloor \tau_1 \to \cdots \to \tau_n \to \tau \rfloor\};\quad f^t = \{\lambda\, x_1 :: \lfloor\tau_1\rfloor.\, \cdots \{\lambda\, x_n :: \lfloor\tau_n\rfloor.\, \lceil e \rceil\} \cdots\}$$

where we make use of the translation functions $\lfloor\cdot\rfloor$ for types and $\lceil\cdot\rceil$ for expressions, defined below. The transition to nested lambda-abstractions allows for the intercalation of set brackets, which produce set-typed terms.

Types are translated from CuMin to SaLT as follows:

$$
\begin{array}{lll}
\lfloor \mathsf{Bool} \rfloor = \mathsf{Bool} & \lfloor \alpha \rfloor = \alpha & \lfloor (\tau, \tau') \rfloor = (\lfloor\tau\rfloor, \lfloor\tau'\rfloor) \\
\lfloor \mathsf{Nat} \rfloor = \mathsf{Nat} & \lfloor [\tau] \rfloor = [\lfloor\tau\rfloor] & \lfloor \tau \to \tau' \rfloor = \lfloor\tau\rfloor \to \{\lfloor\tau'\rfloor\}
\end{array}
$$

The translation function $\lceil\cdot\rceil$ for expressions is shown in Figure 9. The basic idea is to translate every expression of CuMin into a set-typed SaLT expression. Variables, literals and failure are therefore wrapped into singleton sets (which in the case of failure simply means that failure is mapped to failure). All expressions that act on the element level of SaLT are lifted to the set level using indexed unions. This requires the introduction of additional variables with sufficiently fresh names. The anything-primitive already acts on the set level and requires no modification.

The following lemma expresses the expected, and indeed factual, formal properties of the translation and is proved by straightforward inductions.

Lemma 5.1. *Let* $\Gamma = \overline{\alpha_m^{\nu_m}}, \overline{x_n :: \tau_n}$ *be given, and define* $\Gamma' = \overline{\alpha_m^{\nu_m}}, \overline{x_n :: \lfloor\tau_n\rfloor}$.

1. *If* τ *is a CuMin type within* Γ*, then* $\lfloor\tau\rfloor$ *is a SaLT type within* Γ'.
2. *If* $\Gamma \vdash \tau \in \mathsf{Data}$ *holds, then* $\Gamma \vdash \lfloor\tau\rfloor \in \mathsf{Data}$ *holds as well.*
3. *If* $\Gamma \vdash e :: \tau$ *holds as a CuMin typing judgment w.r.t. some program* P*, then* $\Gamma' \vdash \lceil e \rceil :: \{\lfloor\tau\rfloor\}$ *holds as a SaLT typing judgment w.r.t. the translation of* P.
4. *If* θ, σ *are a type and term environment for the typing context* Γ*, then* θ, σ *are also a type and term environment for* Γ'.
5. $[\![\lfloor\tau\rfloor]\!]_\theta = [\![\tau]\!]_\theta$
6. $[\![\lceil e \rceil]\!]^i_{\theta,\sigma} = [\![e]\!]^i_{\theta,\sigma}$
7. $[\![\lceil e \rceil]\!]_{\theta,\sigma} = [\![e]\!]_{\theta,\sigma}$

Before seeing the translation in action, let us remark that in particular it translates the user defined binary choice operator from CuMin to SaLT. While it is not true that $\lceil e_1 \cup e_2 \rceil$ is syntactically exactly $\lceil e_1 \rceil \cup \lceil e_2 \rceil$, these two terms are semantically equivalent in SaLT, and we will freely use that fact as a shortcut during translation. Now, consider again the double coin example in Figure 4. The translation of *double* is $double^t = \{\lambda\, n.\, \{n\} \ni n' \bigcup \{n\} \ni n'' \bigcup \{n' + n''\}\}$ and can be simplified to $double^t = \{\lambda\, n.\, \{n+n\}\}$ using the first monad law twice. Inlining $double^t$ and $coin^t$ into $dc1^t$ (which is a semantics-preserving transformation in SaLT) gives $dc1^t \equiv \{\lambda\, n.\, \{n+n\}\} \ni d \bigcup (\{\{0\} \cup \{1\}) \ni c \bigcup d\ c$. After using the first monad law again and applying beta-reduction, we get $dc1^t \equiv (\{\{0\} \cup \{1\}) \ni c \bigcup \{c + c\}$, and this clearly results in $\{\{0\} \cup \{2\}$. The translation of *dc2* does not allow any simplifications using monad laws. We can again inline $coin^t$ (while *double* has already been inlined on the CuMin side) and get $dc2^t \equiv (\{\{0\} \cup \{1\}) \ni c_1 \bigcup (\{\{0\} \cup \{1\}) \ni c_2 \bigcup \{c_1 + c_2\}$. Since c_1 and c_2 are independent variables, the result is $\{\{0\} \cup \{1\} \cup \{2\}$.

6. Determinism

Our reason for going through the laborious process of introducing SaLT was to make the nondeterminism of CuMin explicit. Now, when we want to investigate some CuMin program, we can instead look at its translation into SaLT. After having cleaned up code by using equational reasoning (specifically formula (3)), some set-typed expressions might be identified as singleton sets. If, for example, we can transform a SaLT function $\Gamma \vdash f :: \tau_1 \to \{\tau_2\}$ into a semantically equivalent composition $\{\_\} \circ \hat{f}$ for some $\hat{f}$ of type $\tau_1 \to \tau_2$, then we know f to be deterministic.

In many cases we can even allow a certain degree of nondeterminism: We call a CuMin function $\Gamma \vdash g :: \tau_1 \to \tau_2$ *multi-deterministic* if there is a SaLT term $\Gamma \vdash \hat{g} :: \{\lfloor\tau_1\rfloor \to \lfloor\tau_2\rfloor\}$ such that

$$\lceil g \rceil \equiv sMap\ (\lambda\, \hat{g}' :: \lfloor\tau_1\rfloor \to \lfloor\tau_2\rfloor.\, \{\_\} \circ \hat{g}')\ \hat{g}$$

All such witnesses $\hat{g}$ for a given g are semantically equivalent.

For a g as above, the translation $\lceil g \rceil$ has the type $\{\lfloor\tau_1\rfloor \to \{\lfloor\tau_2\rfloor\}\}$, and $\hat{g}$ proves the inner level of set brackets to be cosmetic: Because of the syntactic structure (of the semantically equivalent replacement for $\lceil g \rceil$) only singleton sets can occur there. For example, $g = mayInc1$ (see Figure 4) is a multi-deterministic CuMin function, which is witnessed by $\hat{g} = \{\lambda\, x.x\} \cup \{\lambda\, x.x+1\}$:

$$
\begin{array}{rcl}
\lceil g \rceil &\equiv& mayInc1^t \\
&\equiv& id^t_{\mathsf{Nat}} \cup inc^t \\
&\equiv& \{\lambda\, x.\, \{x\}\} \cup \{\lambda\, x.\, \{x+1\}\} \\
&\equiv& \{\{\_\} \circ (\lambda\, x.x)\} \cup \{\{\_\} \circ (\lambda\, x.x+1)\} \\
&\equiv& sMap\ (\{\_\} \circ)\ (\{\lambda\, x.x\} \cup \{\lambda\, x.x+1\})
\end{array}
$$

The translation of *mayInc2*, on the other hand, is a (singleton) set containing a truly nondeterministic function (see Figure 4), and therefore *mayInc2* is not multi-deterministic.

The discussion in [4] shows (by giving counterexamples) that in order for free theorems to hold, the inner level of nondeterminism

has to be restricted, and suggests that the outer level can be tolerated. The formalization of multi-determinism therein – validity of let $y = g\ x$ in $(y,y) \equiv$ let $g' = g$ in let $x' = x$ in $(g'\ x', g'\ x')$ – is implied by the one given above. This can be checked by translating both sides of the semantic equivalence into SaLT and using the witness.

7. Parametricity in SaLT

Since SaLT is essentially a typed lambda-calculus with some additional features, we can, for proving parametricity, rely on existing work and only need to supply the necessary amendments. As usual, establishing parametricity depends on the definition of a logical relation by induction on the syntactic structure of types. Of course, this means that we have to define how to extend the logical relation from an element type to the according set type.

Let P_1 and P_2 be two posets and R a relation between them. The relation $\mathscr{P}_\ell(R)$ between $\mathscr{P}_\ell(P_1)$ and $\mathscr{P}_\ell(P_2)$ can be described like this: Two lower sets $A \in \mathscr{P}_\ell(P_1)$ and $B \in \mathscr{P}_\ell(P_2)$ are related if for every $\mathbf{a} \in A$ there are elements $\mathbf{a}' \in A$ and $\mathbf{b} \in B$ related by R, such that $\mathbf{a} \sqsubseteq \mathbf{a}'$, and analogously for every $\mathbf{b} \in B$. Thus, in order for A and B to be related, it is not necessary for every $\mathbf{a} \in A$ itself to be related to some $\mathbf{b} \in B$[7]. The definition we actually use is equivalent to the above description, though we prefer to state it like this:

$$\begin{aligned}(A,B) \in \mathscr{P}_\ell(R) \iff \exists W \subseteq R.\ & W \neq \emptyset \\ & \wedge\ A = \textstyle\bigcup_{(\mathbf{a},\mathbf{b})\in W} {\downarrow}\mathbf{a} \\ & \wedge\ B = \textstyle\bigcup_{(\mathbf{a},\mathbf{b})\in W} {\downarrow}\mathbf{b}\end{aligned}$$

One possible choice for W is $(A \times B) \cap R$ whenever A and B are related, but other choices might be more useful in proofs at times.

For posets P_1 and P_2 we call a relation R *strict* if $(\bot,\bot) \in R$ and *whole* if $(P_1,P_2) \in \mathscr{P}_\ell(R)$. Strictness is relevant because of the polymorphic failure-primitive, wholeness because of the (restrictedly) polymorphic anything-primitive. The following three lemmas establish some basic properties of $\mathscr{P}_\ell(R)$.

Lemma 7.1. *Let P_1 and P_2 be posets and let $R \subseteq P_1 \times P_2$. If $(\mathbf{x},\mathbf{y}) \in R$, then $({\downarrow}\mathbf{x},{\downarrow}\mathbf{y}) \in \mathscr{P}_\ell(R)$.*

Proof. Indeed $\{(\mathbf{x},\mathbf{y})\} \subseteq R$ with ${\downarrow}\mathbf{x} = \bigcup_{(\mathbf{a},\mathbf{b})\in\{(\mathbf{x},\mathbf{y})\}} {\downarrow}\mathbf{a}$ and accordingly for ${\downarrow}\mathbf{y}$. □

Lemma 7.2. *Let P_1 and P_2 be posets and let $R \subseteq P_1 \times P_2$. Furthermore, let I be some nonempty (index) set, and let $A_i \in \mathscr{P}_\ell(P_1)$ and $B_i \in \mathscr{P}_\ell(P_2)$ for $i \in I$ be two collections of lower sets. If $(A_i,B_i) \in \mathscr{P}_\ell(R)$ for every $i \in I$, then also $(\bigcup_{i\in I} A_i, \bigcup_{i\in I} B_i) \in \mathscr{P}_\ell(R)$.*

Proof. By the definition of $\mathscr{P}_\ell(R)$, there is some nonempty $W_i \subseteq R$ for every $i \in I$ with $A_i = \bigcup_{(\mathbf{a},\mathbf{b})\in W_i} {\downarrow}\mathbf{a}$ and accordingly for B_i. Then $W = \bigcup_{i\in I} W_i \subseteq R$ is nonempty and a witness for $\bigcup_{i\in I} A_i$ and $\bigcup_{i\in I} B_i$ being related, since

$$\bigcup_{i\in I} A_i = \bigcup_{i\in I} \bigcup_{(\mathbf{a},\mathbf{b})\in W_i} {\downarrow}\mathbf{a} = \bigcup_{(\mathbf{a},\mathbf{b})\in W} {\downarrow}\mathbf{a}$$

and accordingly for $\bigcup_{i\in I} B_i$. □

Lemma 7.3. *Let P_1, P_2, Q_1, and Q_2 be posets and let $R \subseteq P_1 \times P_2$ and $S \subseteq Q_1 \times Q_2$. Furthermore, let $(A_1,A_2) \in \mathscr{P}_\ell(R)$ and let $\mathbf{f}_1 : P_1 \to \mathscr{P}_\ell(Q_1)$ and $\mathbf{f}_2 : P_2 \to \mathscr{P}_\ell(Q_2)$ be monotone functions such that $\forall(\mathbf{a}_1,\mathbf{a}_2) \in R.\ (\mathbf{f}_1\ \mathbf{a}_1, \mathbf{f}_2\ \mathbf{a}_2) \in \mathscr{P}_\ell(S)$. Then*

$$\left(\bigcup_{\mathbf{a}_1\in A_1} \mathbf{f}_1\ \mathbf{a}_1,\ \bigcup_{\mathbf{a}_2\in A_2} \mathbf{f}_2\ \mathbf{a}_2\right) \in \mathscr{P}_\ell(S)$$

[7] cf. Lemma 7.1 below, which would be wrong otherwise: ${\downarrow}\mathbf{x}$ can contain elements $\mathbf{a}$ that need not be related to any $\mathbf{b} \in {\downarrow}\mathbf{y}$, even when $\mathbf{x}$ and $\mathbf{y}$ are related.

Proof. By assumption there is a nonempty $U \subseteq R$ with $A_j = \bigcup_{(\mathbf{a}_1,\mathbf{a}_2)\in U} {\downarrow}\mathbf{a}_j$ for $j \in \{1,2\}$. By another assumption, for every $(\mathbf{a}_1,\mathbf{a}_2) \in U$ there is a nonempty $V(\mathbf{a}_1,\mathbf{a}_2) \subseteq S$ with $\mathbf{f}_j\ \mathbf{a}_j = \bigcup_{(\mathbf{b}_1,\mathbf{b}_2)\in V(\mathbf{a}_1,\mathbf{a}_2)} {\downarrow}\mathbf{b}_j$ for $j \in \{1,2\}$. Set $W = \bigcup_{(\mathbf{a}_1,\mathbf{a}_2)\in U} V(\mathbf{a}_1,\mathbf{a}_2) \subseteq S$, which is not empty because neither U nor any $V(\mathbf{a}_1,\mathbf{a}_2)$ is. This W shows the unions to be related, since for $j \in \{1,2\}$:

$$\begin{aligned}\bigcup_{\mathbf{a}_j\in A_j} \mathbf{f}_j\ \mathbf{a}_j &= \bigcup_{(\mathbf{a}'_1,\mathbf{a}'_2)\in U} \bigcup_{\mathbf{a}_j\in{\downarrow}\mathbf{a}'_j} \mathbf{f}_j\ \mathbf{a}_j = \bigcup_{(\mathbf{a}'_1,\mathbf{a}'_2)\in U} \mathbf{f}_j\ \mathbf{a}'_j \\ &= \bigcup_{(\mathbf{a}'_1,\mathbf{a}'_2)\in U} \bigcup_{(\mathbf{b}_1,\mathbf{b}_2)\in V(\mathbf{a}'_1,\mathbf{a}'_2)} {\downarrow}\mathbf{b}_j = \bigcup_{(\mathbf{b}_1,\mathbf{b}_2)\in W} {\downarrow}\mathbf{b}_j\end{aligned}$$

□

Definition 7.4. Let Γ be a typing context and let ρ map every type variable α in Γ to a relation between $\theta_1(\alpha)$ and $\theta_2(\alpha)$ for two type environments θ_1 and θ_2. By induction on the syntactic structure of types, we define for every τ that is a type within Γ, a relation $\Delta_{\rho,\tau}$ between $[\![\tau]\!]_{\theta_1}$ and $[\![\tau]\!]_{\theta_2}$ as given in Figure 10, where Id_P is the identity relation on a poset P.

Lemma 7.5. *If in the situation of Definition 7.4, ρ maps every type variable α in Γ to a strict relation, then the relation $\Delta_{\rho,\tau}$ is strict for every type τ.*

Proof. For Booleans, the naturals, tuples, and lists, this follows from the construction. For type variables, it is an explicit requirement. The only two induction steps to check are for the set type and the function type. By assumption, $\Delta_{\rho,\tau}$ is strict, which means $(\bot,\bot) \in \Delta_{\rho,\tau}$. Using Lemma 7.1 we conclude $({\downarrow}\bot,{\downarrow}\bot) \in \mathscr{P}_\ell\big(\Delta_{\rho,\tau}\big)$, which means $\Delta_{\rho,\{\tau\}}$ is strict, too.

The least element of each function space is the function mapping every argument to $\bot$. And by the definition of $\Delta_{\rho,\tau\to\tau'}$ the constant function to $\bot$ is indeed related to the constant function to $\bot$, since $(\bot,\bot) \in \Delta_{\rho,\tau'}$ by assumption. □

Lemma 7.6. *If in the situation of Definition 7.4, ρ maps every type variable α that is $*$-tagged in Γ to a whole*[8] *relation, and $\Gamma \vdash \tau \in \mathsf{Data}$ holds, then $\Delta_{\rho,\tau}$ is whole.*

Proof. Again by induction. We have to do five cases, corresponding to the five rules for being a data type (cf. Figure 3). The first case is $\Gamma', \alpha^* \vdash \alpha \in \mathsf{Data}$. Since $\Delta_{\rho,\alpha} = \rho(\alpha)$ and α is $*$-tagged, the relation is whole by the assumption on ρ. The cases $\Gamma \vdash \mathsf{Bool} \in \mathsf{Data}$ and $\Gamma \vdash \mathsf{Nat} \in \mathsf{Data}$ are trivial since the relation $\Delta_{\rho,\tau}$ is the identity relation in both cases. In the inductive case

$$\frac{\Gamma \vdash \tau \in \mathsf{Data}}{\Gamma \vdash [\tau] \in \mathsf{Data}}$$

we assume $([\![\tau]\!]_{\theta_1}, [\![\tau]\!]_{\theta_2}) \in \mathscr{P}_\ell\big(\Delta_{\rho,\tau}\big)$ and want to conclude that $([\![[\tau]]\!]_{\theta_1}, [\![[\tau]]\!]_{\theta_2}) \in \mathscr{P}_\ell\Big(\Delta_{\rho,[\tau]}\Big)$. By the assumption there is a nonempty $U \subseteq \Delta_{\rho,\tau}$ with $[\![\tau]\!]_{\theta_1} = \bigcup_{(\mathbf{x},\mathbf{y})\in U} {\downarrow}\mathbf{x}$ and accordingly for $[\![\tau]\!]_{\theta_2}$. Set $W = \{(\mathbf{x}_1 : \cdots : \mathbf{x}_n : [], \mathbf{y}_1 : \cdots : \mathbf{y}_n : []) \mid n \geq 0, (\mathbf{x}_i,\mathbf{y}_i) \in U\} \subseteq \Delta_{\rho,[\tau]}$, which obviously is nonempty. This W shows the desired conclusion, since

$$\begin{aligned}[\![[\tau]]\!]_{\theta_1} &= \{\mathbf{x}_1 : \cdots : \mathbf{x}_n : \mathbf{e} \mid n \geq 0, \mathbf{x}_i \in [\![\tau]\!]_{\theta_1}, \mathbf{e} \in \{\bot,[]\}\} \\ &= \bigcup_{n\geq 0} \bigcup_{((\mathbf{x}'_1,\mathbf{y}'_1),\ldots,(\mathbf{x}'_n,\mathbf{y}'_n))\in U^n} \{\mathbf{x}_1 : \cdots : \mathbf{x}_n : \mathbf{e} \mid \mathbf{x}_i \in {\downarrow}\mathbf{x}'_i, \\ &\qquad\qquad \mathbf{e} \in \{\bot,[]\}\} \\ &= \bigcup_{n\geq 0} \bigcup_{((\mathbf{x}'_1,\mathbf{y}'_1),\ldots,(\mathbf{x}'_n,\mathbf{y}'_n))\in U^n} {\downarrow}(\mathbf{x}'_1 : \cdots : \mathbf{x}'_n : []) \\ &= \bigcup_{(\mathbf{a}_1,\mathbf{a}_2)\in W} {\downarrow}\mathbf{a}_1\end{aligned}$$

[8] Recall the definition from earlier in this section.

$$\Delta_{\rho,\alpha} = \rho(\alpha) \qquad \Delta_{\rho,\mathsf{Bool}} = Id_{[\![\mathsf{Bool}]\!]_\emptyset} \qquad \Delta_{\rho,\tau\to\tau'} = \{(\mathbf{f},\mathbf{g}) \in [\![\tau\to\tau']\!]_{\theta_1} \times [\![\tau\to\tau']\!]_{\theta_2} \mid \forall(\mathbf{x},\mathbf{y}) \in \Delta_{\rho,\tau}.\,(\mathbf{f}\,\mathbf{x},\mathbf{g}\,\mathbf{y}) \in \Delta_{\rho,\tau'}\}$$
$$\Delta_{\rho,\{\tau\}} = \mathscr{P}_\ell(\Delta_{\rho,\tau}) \qquad \Delta_{\rho,\mathsf{Nat}} = Id_{[\![\mathsf{Nat}]\!]_\emptyset} \qquad \Delta_{\rho,[\tau]} = \{(\mathbf{x}_1 : \cdots : \mathbf{x}_n : \mathbf{e}, \mathbf{y}_1 : \cdots : \mathbf{y}_n : \mathbf{e}) \mid n \geq 0, (\mathbf{x}_i,\mathbf{y}_i) \in \Delta_{\rho,\tau}, \mathbf{e} \in \{\bot,[]\}\}$$
$$\Delta_{\rho,(\tau,\tau')} = \{(\bot,\bot)\} \cup \{((\mathbf{l}_1,\mathbf{r}_1),(\mathbf{l}_2,\mathbf{r}_2)) \mid (\mathbf{l}_1,\mathbf{l}_2) \in \Delta_{\rho,\tau}, (\mathbf{r}_1,\mathbf{r}_2) \in \Delta_{\rho,\tau'}\}$$

Figure 10. Defining equations for the logical relation

and accordingly for $[\![[\tau]]\!]_{\theta_2}$. The last case, also inductive, is:

$$\frac{\Gamma \vdash \tau \in \mathsf{Data} \qquad \Gamma \vdash \tau' \in \mathsf{Data}}{\Gamma \vdash (\tau,\tau') \in \mathsf{Data}}$$

Its proof is analogous to that for the list case above, using

$$\{\bot\} \cup \bigcup_{(\mathbf{x}'_1,\mathbf{y}'_1)\in U_1} \bigcup_{(\mathbf{x}'_2,\mathbf{y}'_2)\in U_2} \{(\mathbf{x}_1,\mathbf{x}_2) \mid \mathbf{x}_1 \in {\downarrow}\mathbf{x}'_1, \mathbf{x}_2 \in {\downarrow}\mathbf{x}'_2\}$$
$$= \bigcup_{(\mathbf{x}'_1,\mathbf{y}'_1)\in U_1} \bigcup_{(\mathbf{x}'_2,\mathbf{y}'_2)\in U_2} {\downarrow}(\mathbf{x}'_1,\mathbf{x}'_2)$$

as the crucial step for the statement concerning $[\![(\tau,\tau')]\!]_{\theta_1}$, and accordingly with $\mathbf{y}$ for the statement concerning $[\![(\tau,\tau')]\!]_{\theta_2}$. □

Note that Lemma 7.6, which is crucial for dealing with the anything-primitive in the proof of the following theorem, would not hold if one would allow non-whole relations for $*$-tagged α or if function types were allowed in Data. In fact, the parametricity theorem would not hold without the decisions made concerning type variable tagging and Data. To avoid the tagging, we could of course simply require whole relations for *all* type variables, but that would considerably weaken the power of parametricity and free theorems in situations where the anything-primitive is not, or sparingly, used.

Theorem 7.7 (Parametricity for SaLT)**.** *Let $\Gamma \vdash e :: \tau$ (in SaLT) be valid w.r.t. some program P and let θ_1,σ_1 and θ_2,σ_2 be appropriate pairs of type and term environments. Let ρ be given as in Definition 7.4 and let it map every type variable to a strict relation that is also whole for type variables that are $*$-tagged in Γ. If for all $(x :: \tau') \in \Gamma$ we have $(\sigma_1(x),\sigma_2(x)) \in \Delta_{\rho,\tau'}$, then also $([\![e]\!]^i_{\theta_1,\sigma_1},[\![e]\!]^i_{\theta_2,\sigma_2}) \in \Delta_{\rho,\tau}$ for all $i \in \mathbb{N}$. If additionally τ is a set type, then $([\![e]\!]_{\theta_1,\sigma_1},[\![e]\!]_{\theta_2,\sigma_2}) \in \Delta_{\rho,\tau}$.*

Proof. The claim concerning $[\![e]\!]_{\theta_1,\sigma_1}$ and $[\![e]\!]_{\theta_2,\sigma_2}$ follows directly from applying Lemma 7.2 to the claim concerning $[\![e]\!]^i_{\theta_1,\sigma_1}$ and $[\![e]\!]^i_{\theta_2,\sigma_2}$. We show the latter claim by two nested layers of induction. The outer layer is induction on i and the inner layer is induction on the structure of the expression e. We split cases according to the syntax, and for every syntactic construct we assume the claim to be true for all subexpressions, thus making use of the inner layer induction hypothesis.

Most of the syntax is already present in standard deterministic lambda-calculi and the proof for these cases can be found in the literature. Here the case of function application shall serve as an example of how individual cases are done. The typing rule is

$$\frac{\Gamma \vdash e_1 :: \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 :: \tau_1}{\Gamma \vdash e_1\ e_2 :: \tau_2}$$

so we may assume $([\![e_2]\!]^i_{\theta_1,\sigma_1},[\![e_2]\!]^i_{\theta_2,\sigma_2}) \in \Delta_{\rho,\tau_1}$ by the induction hypothesis for the right premise. For the left premise we get $([\![e_1]\!]^i_{\theta_1,\sigma_1},[\![e_1]\!]^i_{\theta_2,\sigma_2}) \in \Delta_{\rho,\tau_1\to\tau_2}$, which by the definition of the logical relation for function types means that for all $(\mathbf{x},\mathbf{y}) \in \Delta_{\rho,\tau_1}$ also $([\![e_1]\!]^i_{\theta_1,\sigma_1}\ \mathbf{x},[\![e_1]\!]^i_{\theta_2,\sigma_2}\ \mathbf{y}) \in \Delta_{\rho,\tau_2}$. Thus in particular $([\![e_1]\!]^i_{\theta_1,\sigma_1}\ [\![e_2]\!]^i_{\theta_1,\sigma_1},[\![e_1]\!]^i_{\theta_2,\sigma_2}\ [\![e_2]\!]^i_{\theta_2,\sigma_2}) \in \Delta_{\rho,\tau_2}$, which is equivalent to the claim $([\![e_1\ e_2]\!]^i_{\theta_1,\sigma_1},[\![e_1\ e_2]\!]^i_{\theta_2,\sigma_2}) \in \Delta_{\rho,\tau_2}$ by a definition.

The case for invoking a function symbol with function definition $f = rhs$ splits into two subcases distinguishing whether i is zero or positive. If it is zero, we are in the base case of the outer layer of induction and the semantics of $f_{\overline{\tau_m}}$ is $\bot$ both w.r.t. $[\![\cdot]\!]^0_{\theta_1,\sigma_1}$ and $[\![\cdot]\!]^0_{\theta_2,\sigma_2}$. Because of Lemma 7.5, $\bot$ is related to itself. Otherwise, the semantics of $f_{\overline{\tau_m}}$ (both w.r.t. $[\![\cdot]\!]^i_{\theta_1,\sigma_1}$ and $[\![\cdot]\!]^i_{\theta_2,\sigma_2}$) is defined via the semantics of *rhs* at the lower index $i-1$ and we can use the outer layer induction hypothesis. In order to make sure the relation environment $[\overline{\alpha_m \mapsto \Delta_{\rho,\tau_m}}]$ is appropriate in the induction hypothesis, one has to check that type variables which are $*$-tagged in f's type signature are mapped to whole relations, which is the case by Lemma 7.6.

All remaining cases, for the primitives involving set types, are covered by the lemmas we have shown before: Singleton sets are covered by Lemma 7.1, indexed unions by Lemma 7.3, and the anything-primitive by Lemma 7.6. □

The conclusion of Theorem 7.7 is that the same expression evaluated in two different semantic environments gives two related semantic values. This can be hard to work with since being related is a rather implicit notion. The standard way to deal with this is to unravel the definition of being related, in order to get semantic equivalence of two different expressions. Another way of harnessing Theorem 7.7 is presented in the following theorem, which proves semantic equivalence of two expressions directly. Here $id_\tau = \lambda\, x :: \tau.x$ is an identity function term in SaLT.

Theorem 7.8. *Let Γ be some typing context for SaLT such that τ, τ_1, and τ_2 are types within Γ, and let Γ' be an extension of Γ to Γ,α^v, in $:: \tau_1 \to \alpha$, out $:: \alpha \to \tau_2$. Let $\Gamma \vdash e :: \tau_1 \to \tau_2$ and $\Gamma' \vdash e' :: \tau$ be valid SaLT typing judgments. If e $\mathsf{failure}_{\tau_1} \equiv \mathsf{failure}_{\tau_2}$, and in case of $*$-tagged α also $\Gamma \vdash \tau_1 \in \mathsf{Data}$, $\Gamma \vdash \tau_2 \in \mathsf{Data}$, and sMap e $\mathsf{anything}_{\tau_1} \equiv \mathsf{anything}_{\tau_2}$ hold, then*

$$e'[\tau_1/\alpha, id_{\tau_1}/in, e/out] \equiv e'[\tau_2/\alpha, e/in, id_{\tau_2}/out]$$

When this theorem is applied, it looks as though the subexpression e was transported from one place to another. Yet the two placeholders *in* and *out* are there all the time and only take turns in being replaced by e. Also, we do not have to commit ourselves as to what *the* free theorem for some type is. As long as we can construct the more general expression e' using the additional type variable α and the two placeholders *in* and *out*, we are free to choose.

SaLT terms e that satisfy e $\mathsf{failure}_{\tau_1} \equiv \mathsf{failure}_{\tau_2}$ are called *strict*. A SaLT term e is called *onto* if *sMap* e $\mathsf{anything}_{\tau_1} \equiv \mathsf{anything}_{\tau_2}$ holds. The semantics of e does not really have to produce every single value though. It is sufficient that for every value $\mathbf{y}$ there is some $\mathbf{y}' \sqsupseteq \mathbf{y}$ in the image; such functions are called *final*.

Let us show an example use of Theorem 7.8, namely proving the extensionality property that the type $\forall\alpha.\,\alpha \to \{\alpha\}$ is only inhabited by failure and $\{\_\}$. Let be given a SaLT program containing a function symbol f with type annotation $f :: \forall\alpha.\,\alpha \to \{\alpha\}$. We pick some closed type τ_2 and term x of that type, and invoke Theorem 7.8 with $e = \lambda\, b.\,\mathsf{case}\ b\ \mathsf{of}\ \langle \mathsf{True} \to x; \mathsf{False} \to x\rangle$ (which is strict) and $e' = sMap\ out\ (f_\alpha\ (in\ \mathsf{True}))$ (which fixes τ_1 to Bool and τ to $\{\tau_2\}$). We get the semantic equivalence *sMap* e $(f_{\mathsf{Bool}}\ (id_{\mathsf{Bool}}\ \mathsf{True})) \equiv$ *sMap* id_{τ_2} $(f_{\tau_2}\ (e\ \mathsf{True}))$, from which it is easy to see that $f_{\tau_2}\ x \equiv$ *sMap* e $(f_{\mathsf{Bool}}\ \mathsf{True})$. Now let us consider the term $f_{\mathsf{Bool}}\ \mathsf{True}$, which is of type $\{\mathsf{Bool}\}$. That is not a very rich type. Indeed, $f_{\mathsf{Bool}}\ \mathsf{True}$ must be semantically equivalent to one of the following:

$\mathsf{failure}_{\{\mathsf{Bool}\}}$, $\{\mathsf{True}\}$, $\{\mathsf{False}\}$, $\mathsf{anything}_{\mathsf{Bool}}$. Obviously, to *which* of those four it is equivalent, is independent of the choice of τ_2 and x above. Thus, using the semantics of *sMap* and *e*, we obtain that either for every τ_2 and x, $f_{\tau_2}\ x \equiv \mathsf{failure}_{\{\tau_2\}}$, or for every τ_2 and x, $f_{\tau_2}\ x \equiv \{x\}$.

We do not give the proof of Theorem 7.8 here. It is largely shared with the proof of Lemma 8.1 given in the next section.

8. Toward Deriving Free Theorems for CuMin

For simplifying the use of SaLT parametricity in establishing free theorems for CuMin programs, we prepare a theorem (Theorem 8.3 below) similar in spirit to Theorem 7.8 but tailored to specific situations that occur when working with SaLT translations of desired CuMin equivalences. First, we discuss CuMin analogues of the notions *strict* and *onto*.

We call a CuMin term $\Gamma \vdash g :: \tau_1 \to \tau_2$ *strict* if $g\ \mathsf{failure}_{\tau_1} \equiv \mathsf{failure}_{\tau_2}$. As in SaLT, $\equiv$ means having equal semantics for arbitrary environments. If $\Gamma \vdash \tau_1 \in \mathsf{Data}$ and $\Gamma \vdash \tau_2 \in \mathsf{Data}$, we call *g multi-onto* if it is multi-deterministic and any SaLT term $\hat{g}$ witnessing this (remember, all such witnesses are semantically equivalent to each other) satisfies the following SaLT equivalence:

$$\begin{aligned} & \hat{g} \ni \hat{g}' \bigcup \{(\hat{g}', \mathit{sMap}\ \hat{g}'\ \mathsf{anything}_{\lfloor\tau_1\rfloor})\} \\ \equiv\ & \hat{g} \ni \hat{g}' \bigcup \{(\hat{g}', \mathsf{anything}_{\lfloor\tau_2\rfloor})\} \end{aligned}$$

A good intuition for this is seeing $\hat{g}$ as a set of surjective functions, though actually finality instead of surjectivity (of the constituents of the semantics of $\hat{g}$) is sufficient, as before. Also, not every single member/constituent has to be final, but for every one there has to be a more or equally defined final one. However, it is not sufficient that all member functions together produce every value in the target. For example, the multi-deterministic CuMin function $\mathit{mayInc1} = id_{\mathsf{Nat}} \cup \mathit{inc}$ is not multi-onto, since for $\hat{g} = \{id_{\mathsf{Nat}}\} \cup \{\lambda x :: \mathsf{Nat}.x+1\}$ (and thus for every one of the all semantically equivalent witnesses) the left-hand side of the above supposed equivalence will lack any pair with left component the function $\lambda x :: \mathsf{Nat}.x+1$ and right component a set containing 0.

In the following, we write $G_{\mathbf{f}} = \{(\mathbf{x}, \mathbf{f}\ \mathbf{x}) \mid \mathbf{x} \in A\}$ for the graph of a function $\mathbf{f} : A \to B$.

Lemma 8.1. *Let Γ be some typing context for SaLT such that τ, τ_1, and τ_2 are types within Γ, and let Γ' be an extension of Γ to $\Gamma, \alpha^{\nu}, \mathit{in} :: \tau_1 \to \alpha, \mathit{out} :: \alpha \to \tau_2$. Let $\Gamma \vdash e :: \tau_1 \to \tau_2$ and $\Gamma' \vdash e' :: \tau$ be valid SaLT typing judgments, let θ, σ be some environment pair appropriate for Γ, and let $i \in \mathbb{N}$. If $G_{[\![e]\!]^i_{\theta,\sigma}}$ is strict, and in case of $*$-tagged α also whole (and assuming in that case that additionally $\Gamma \vdash \tau_1 \in \mathsf{Data}$ and $\Gamma \vdash \tau_2 \in \mathsf{Data}$ hold), then*

$$[\![e'[\tau_1/\alpha, id_{\tau_1}/\mathit{in}, e/\mathit{out}]]\!]^i_{\theta,\sigma} = [\![e'[\tau_2/\alpha, e/\mathit{in}, id_{\tau_2}/\mathit{out}]]\!]^i_{\theta,\sigma}$$

Proof. Define $\theta_1 = \theta[\alpha \mapsto [\![\tau_1]\!]_\theta]$, $\theta_2 = \theta[\alpha \mapsto [\![\tau_2]\!]_\theta]$, $\sigma_1 = \sigma[\mathit{in} \mapsto id_{[\![\tau_1]\!]_\theta}, \mathit{out} \mapsto [\![e]\!]^i_{\theta,\sigma}]$, and $\sigma_2 = \sigma[\mathit{in} \mapsto [\![e]\!]^i_{\theta,\sigma}, \mathit{out} \mapsto id_{[\![\tau_2]\!]_\theta}]$. (Here *id* is used for semantic functions, not terms.) Let ρ map every type variable in Γ to an identity relation (at the type prescribed by θ) and α to $G_{[\![e]\!]^i_{\theta,\sigma}}$. Every type variable is mapped to an appropriate relation since identity relations are strict and whole and the desired properties for $G_{[\![e]\!]^i_{\theta,\sigma}}$ are premises. An easy induction proof shows that $\Delta_{\rho,\tau'}$ is an identity relation for every τ' that is a type within Γ, since Γ does not contain α. For all term variables $x :: \tau'$ in Γ, we have $\sigma_1(x) = \sigma_2(x) \in [\![\tau']\!]_\theta$ and therefore $(\sigma_1(x), \sigma_2(x)) \in \Delta_{\rho,\tau'}$. Also,

$$\begin{aligned} & (\sigma_1(\mathit{in}), \sigma_2(\mathit{in})) \in \Delta_{\rho,\tau_1\to\alpha} \\ \iff\ & (id_{[\![\tau_1]\!]_\theta}, [\![e]\!]^i_{\theta,\sigma}) \in \Delta_{\rho,\tau_1\to\alpha} \\ \iff\ & \forall (\mathbf{x}_1, \mathbf{x}_2) \in \Delta_{\rho,\tau_1}.\, (\mathbf{x}_1, [\![e]\!]^i_{\theta,\sigma}\ \mathbf{x}_2) \in \Delta_{\rho,\alpha} \\ \iff\ & \forall \mathbf{x} \in [\![\tau_1]\!]_\theta.\, (\mathbf{x}, [\![e]\!]^i_{\theta,\sigma}\ \mathbf{x}) \in G_{[\![e]\!]^i_{\theta,\sigma}} \end{aligned}$$

is true by the definition of Δ and G, and a similar argument shows $(\sigma_1(\mathit{out}), \sigma_2(\mathit{out})) \in \Delta_{\rho,\alpha\to\tau_2}$. Thus, by Theorem 7.7 we conclude $([\![e']\!]^i_{\theta_1,\sigma_1}, [\![e']\!]^i_{\theta_2,\sigma_2}) \in \Delta_{\rho,\tau}$. Since τ is a type within Γ, the relation $\Delta_{\rho,\tau}$ is the identity relation on $[\![\tau]\!]_\theta$. Thus, as desired:

$$\begin{aligned} [\![e'[\tau_1/\alpha, id_{\tau_1}/\mathit{in}, e/\mathit{out}]]\!]^i_{\theta,\sigma} &= [\![e']\!]^i_{\theta_1,\sigma_1} = [\![e']\!]^i_{\theta_2,\sigma_2} \\ &= [\![e'[\tau_2/\alpha, e/\mathit{in}, id_{\tau_2}/\mathit{out}]]\!]^i_{\theta,\sigma} \end{aligned}$$

□

Lemma 8.2. *Let $\mathbf{g} : P_1 \to P_2$ be a function between two posets. If $P_2 = \bigcup_{\mathbf{x}\in P_1} {\downarrow}(\mathbf{g}\ \mathbf{x})$, then $G_{\mathbf{g}}$ is whole.*

Proof. We need to show $(P_1, P_2) \in \mathscr{P}_\ell\big(G_{\mathbf{g}}\big)$. This means finding a nonempty $W \subseteq G_{\mathbf{g}}$ such that $P_1 = \bigcup_{(\mathbf{x},\mathbf{y})\in W} {\downarrow}\mathbf{x}$ and $P_2 = \bigcup_{(\mathbf{x},\mathbf{y})\in W} {\downarrow}\mathbf{y}$. Choosing $W = G_{\mathbf{g}} = \{(\mathbf{x}, \mathbf{g}\ \mathbf{x}) \mid \mathbf{x} \in P_1\}$ works just fine. □

Theorem 8.3. *Let τ_1 and τ_2 be closed CuMin types (i.e., types not containing any type variables). Let τ be a closed SaLT type and let*

$$\alpha^{\nu}, \mathit{in} :: \lfloor\tau_1\rfloor \to \alpha, \mathit{out} :: \alpha \to \lfloor\tau_2\rfloor \vdash e' :: \{\tau\}$$

be a valid SaLT typing judgment. Let $g :: \tau_1 \to \tau_2$ be a strict and multi-deterministic CuMin term that in the case of $*$-tagged α is also multi-onto, and let $\hat{g} :: \{\lfloor\tau_1\rfloor \to \lfloor\tau_2\rfloor\}$ be a witness for g being multi-deterministic and possibly multi-onto. Then the following SaLT equivalence holds:

$$\begin{aligned} & \hat{g} \ni \hat{g}' \bigcup e'[\lfloor\tau_1\rfloor/\alpha, id_{\lfloor\tau_1\rfloor}/\mathit{in}, \hat{g}'/\mathit{out}] \\ \equiv\ & \hat{g} \ni \hat{g}' \bigcup e'[\lfloor\tau_2\rfloor/\alpha, \hat{g}'/\mathit{in}, id_{\lfloor\tau_2\rfloor}/\mathit{out}] \end{aligned}$$

In the case of $*$-tagged α, the theorem implicitly assumes $\vdash \tau_1 \in \mathsf{Data}$ and $\vdash \tau_2 \in \mathsf{Data}$, as the notion "multi-onto" would not apply otherwise.

Proof. We first concentrate on the case that α is not $*$-tagged. We fix some $i \in \mathbb{N}$, some environments θ, σ, and some $\hat{\mathbf{g}}' \in [\![\hat{g}]\!]^i_{\theta,\sigma}$. Then

$$\begin{aligned} \hat{\mathbf{g}}'\ \bot &= [\![\hat{g}'\ \mathsf{failure}_{\lfloor\tau_1\rfloor}]\!]^i_{[\hat{g}'\mapsto\hat{\mathbf{g}}']} \\ &\in {\downarrow}[\![\hat{g}'\ \mathsf{failure}_{\lfloor\tau_1\rfloor}]\!]^i_{[\hat{g}'\mapsto\hat{\mathbf{g}}']} \\ &\subseteq \bigcup_{\hat{\mathbf{g}}''\in[\![\hat{g}]\!]^i_{\theta,\sigma}} {\downarrow}[\![\hat{g}'\ \mathsf{failure}_{\lfloor\tau_1\rfloor}]\!]^i_{[\hat{g}'\mapsto\hat{\mathbf{g}}'']} \\ &= [\![\hat{g} \ni \hat{g}' \bigcup \{\hat{g}'\ \mathsf{failure}_{\lfloor\tau_1\rfloor}\}]\!]^i_{\theta,\sigma} \\ &\subseteq [\![\hat{g} \ni \hat{g}' \bigcup \{\hat{g}'\ \mathsf{failure}_{\lfloor\tau_1\rfloor}\}]\!]_{\theta,\sigma} \\ &= [\![\mathit{sMap}\ (\{\_\}\circ)\ \hat{g} \ni \hat{g}' \bigcup \hat{g}'\ \mathsf{failure}_{\lfloor\tau_1\rfloor}]\!]_{\theta,\sigma} \\ &= [\![\lceil g\rceil \ni \hat{g}' \bigcup \hat{g}'\ \mathsf{failure}_{\lfloor\tau_1\rfloor}]\!]_{\theta,\sigma} \\ &= [\![\lceil g\ \mathsf{failure}_{\tau_1}\rceil]\!]_{\theta,\sigma} \\ &= [\![g\ \mathsf{failure}_{\tau_1}]\!]_{\theta,\sigma} \\ &= [\![\mathsf{failure}_{\tau_2}]\!]_{\theta,\sigma} \\ &= \{\bot\} \end{aligned}$$

and so $\hat{\mathbf{g}}'\ \bot = \bot$, making $G_{\hat{\mathbf{g}}'}$ a strict relation. Then we can use Lemma 8.1 for the judgments

$$\alpha^{\nu}, \mathit{in} :: \lfloor\tau_1\rfloor \to \alpha, \mathit{out} :: \alpha \to \lfloor\tau_2\rfloor, \hat{g}' :: \lfloor\tau_1\rfloor \to \lfloor\tau_2\rfloor \vdash e' :: \{\tau\}$$

and $\hat{g}' :: \lfloor\tau_1\rfloor \to \lfloor\tau_2\rfloor \vdash \hat{g}' :: \lfloor\tau_1\rfloor \to \lfloor\tau_2\rfloor$ and the environments $\theta, \sigma[\hat{g}' \to \hat{\mathbf{g}}']$, to conclude:

$$\begin{aligned} & [\![e'[\lfloor\tau_1\rfloor/\alpha, id_{\lfloor\tau_1\rfloor}/\mathit{in}, \hat{g}'/\mathit{out}]]\!]^i_{\theta,\sigma[\hat{g}'\to\hat{\mathbf{g}}']} \\ =\ & [\![e'[\lfloor\tau_2\rfloor/\alpha, \hat{g}'/\mathit{in}, id_{\lfloor\tau_2\rfloor}/\mathit{out}]]\!]^i_{\theta,\sigma[\hat{g}'\to\hat{\mathbf{g}}']} \end{aligned}$$

If instead of fixing one $i \in \mathbb{N}$ and one $\hat{\mathbf{g}}' \in [\![\hat{g}]\!]^i_{\theta,\sigma}$ we build the unions over all $i \in \mathbb{N}$ and all $\hat{\mathbf{g}}' \in [\![\hat{g}]\!]^i_{\theta,\sigma}$, we get

$$\bigcup_{i\in\mathbb{N}} \bigcup_{\hat{\mathbf{g}}'\in[\![\hat{g}]\!]^i_{\theta,\sigma}} [\![e'[\lfloor\tau_1\rfloor/\alpha, id_{\lfloor\tau_1\rfloor}/in, \hat{g}'/out]]\!]^i_{\theta,\sigma[\hat{g}'\mapsto\hat{\mathbf{g}}']}$$
$$= \bigcup_{i\in\mathbb{N}} \bigcup_{\hat{\mathbf{g}}'\in[\![\hat{g}]\!]^i_{\theta,\sigma}} [\![e'[\lfloor\tau_2\rfloor/\alpha, \hat{g}'/in, id_{\lfloor\tau_2\rfloor}/out]]\!]^i_{\theta,\sigma[\hat{g}'\mapsto\hat{\mathbf{g}}']}$$

which is equivalent to:

$$[\![\hat{g} \ni \hat{g}' \cup e'[\lfloor\tau_1\rfloor/\alpha, id_{\lfloor\tau_1\rfloor}/in, \hat{g}'/out]]\!]_{\theta,\sigma}$$
$$= [\![\hat{g} \ni \hat{g}' \cup e'[\lfloor\tau_2\rfloor/\alpha, \hat{g}'/in, id_{\lfloor\tau_2\rfloor}/out]]\!]_{\theta,\sigma}$$

Since this is true for arbitrary θ, σ, we are done in this case.

In the case that α is $*$-tagged, we can use the additional premise to get:

$$[\![\hat{g} \ni \hat{g}' \cup \{(\hat{g}', sMap\ \hat{g}'\ \mathsf{anything}_{\lfloor\tau_1\rfloor})\}]\!]_{\theta,\sigma}$$
$$= [\![\hat{g} \ni \hat{g}' \cup \{(\hat{g}', \mathsf{anything}_{\lfloor\tau_2\rfloor})\}]\!]_{\theta,\sigma}$$

The right-hand side equals

$$\bigcup_{i\in\mathbb{N}} \bigcup_{\hat{\mathbf{g}}'\in[\![\hat{g}]\!]^i_{\theta,\sigma}} [\![\{(\hat{g}', \mathsf{anything}_{\lfloor\tau_2\rfloor})\}]\!]^i_{\theta,\sigma[\hat{g}'\mapsto\hat{\mathbf{g}}']}$$
$$= \bigcup_{i\in\mathbb{N}} \bigcup_{\hat{\mathbf{g}}'\in[\![\hat{g}]\!]^i_{\theta,\sigma}} \downarrow[\![(\hat{g}', \mathsf{anything}_{\lfloor\tau_2\rfloor})]\!]^i_{\theta,\sigma[\hat{g}'\mapsto\hat{\mathbf{g}}']}$$
$$= \bigcup_{i\in\mathbb{N}} \bigcup_{\hat{\mathbf{g}}'\in[\![\hat{g}]\!]^i_{\theta,\sigma}} \downarrow([\![\hat{g}']\!]^i_{\theta,\sigma[\hat{g}'\mapsto\hat{\mathbf{g}}']}, [\![\mathsf{anything}_{\lfloor\tau_2\rfloor}]\!]^i_{\theta,\sigma[\hat{g}'\mapsto\hat{\mathbf{g}}']})$$
$$= \bigcup_{i\in\mathbb{N}} \bigcup_{\hat{\mathbf{g}}'\in[\![\hat{g}]\!]^i_{\theta,\sigma}} \downarrow(\hat{\mathbf{g}}', [\![\mathsf{anything}_{\lfloor\tau_2\rfloor}]\!]^i_{\theta,\sigma[\hat{g}'\mapsto\hat{\mathbf{g}}']})$$

so if we have some $\hat{\mathbf{g}}' \in [\![\hat{g}]\!]^i_{\theta,\sigma}$, then $(\hat{\mathbf{g}}', [\![\lfloor\tau_2\rfloor]\!]_\theta)$ is an element of that right-hand side. Therefore, it also has to be an element of the original left-hand side, which (by the same steps) is equal to:

$$\bigcup_{j\in\mathbb{N}} \bigcup_{\hat{\mathbf{g}}''\in[\![\hat{g}]\!]^j_{\theta,\sigma}} \downarrow(\hat{\mathbf{g}}'', [\![sMap\ \hat{g}'\ \mathsf{anything}_{\lfloor\tau_1\rfloor}]\!]^j_{\theta,\sigma[\hat{g}'\mapsto\hat{\mathbf{g}}'']})$$

Thus, for every $\hat{\mathbf{g}}' \in [\![\hat{g}]\!]^i_{\theta,\sigma}$, there has to be some $j \in \mathbb{N}$ and a $\hat{\mathbf{g}}'' \in [\![\hat{g}]\!]^j_{\theta,\sigma}$ such that:

$$(\hat{\mathbf{g}}', [\![\lfloor\tau_2\rfloor]\!]_\theta) \sqsubseteq (\hat{\mathbf{g}}'', [\![sMap\ \hat{g}'\ \mathsf{anything}_{\lfloor\tau_1\rfloor}]\!]^j_{\theta,\sigma[\hat{g}'\mapsto\hat{\mathbf{g}}'']})$$

So $\hat{\mathbf{g}}' \sqsubseteq \hat{\mathbf{g}}''$ and $[\![\lfloor\tau_2\rfloor]\!]_\theta = [\![sMap\ \hat{g}'\ \mathsf{anything}_{\lfloor\tau_1\rfloor}]\!]^j_{\theta,\sigma[\hat{g}'\mapsto\hat{\mathbf{g}}'']}$, since $[\![\lfloor\tau_2\rfloor]\!]_\theta$ is the greatest element of $[\![\{\lfloor\tau_2\rfloor\}]\!]_\theta$. The equation can serve as the premise of Lemma 8.2, to prove that $G_{\hat{\mathbf{g}}''}$ is a whole relation.

Now we apply Lemma 8.1 for all $i \in \mathbb{N}$ as before, but this time – since α is $*$-tagged – only for all $\hat{\mathbf{g}}'' \in [\![\hat{g}]\!]^i_{\theta,\sigma}$ with $G_{\hat{\mathbf{g}}''}$ whole (in addition to being strict, which is the case for any element of $[\![\hat{g}]\!]^i_{\theta,\sigma}$ anyway, as we have seen before), to get:

$$[\![e'[\lfloor\tau_1\rfloor/\alpha, id_{\lfloor\tau_1\rfloor}/in, \hat{g}'/out]]\!]^i_{\theta,\sigma[\hat{g}'\to\hat{\mathbf{g}}'']}$$
$$= [\![e'[\lfloor\tau_2\rfloor/\alpha, \hat{g}'/in, id_{\lfloor\tau_2\rfloor}/out]]\!]^i_{\theta,\sigma[\hat{g}'\to\hat{\mathbf{g}}'']}$$

We then, on both sides, build the unions over all $i \in \mathbb{N}$ and all $\hat{\mathbf{g}}'' \in [\![\hat{g}]\!]^i_{\theta,\sigma}$ with $G_{\hat{\mathbf{g}}''}$ whole. Unlike in the case where α is not $*$-tagged, the resulting equation is not immediately equivalent to the desired equation, because of the wholeness restriction affecting the choices for $\hat{\mathbf{g}}''$. But we know from above that for every $\hat{\mathbf{g}}' \in [\![\hat{g}]\!]^i_{\theta,\sigma}$ there is some j and some $\hat{\mathbf{g}}'' \in [\![\hat{g}]\!]^j_{\theta,\sigma}$ with $\hat{\mathbf{g}}' \sqsubseteq \hat{\mathbf{g}}''$ and $G_{\hat{\mathbf{g}}''}$ whole, thus already taking part in the unions on either side of the equation. Because of the monotonicity of the semantics, the unions do not change if $\hat{\mathbf{g}}'$ itself takes part as well, since it does not contribute anything additional anyway. Therefore, we do after all get the desired equation as in the case where α is not $*$-tagged. □

9. Examples of Free Theorems for CuMin

In this section, we deduce four, mostly instructive, free theorems highlighting different aspects of our machinery. A common theme is that we benefit from standard equational reasoning once we are on the SaLT side. In each case, we start from a function (or simply value, in the case of the third example) of which we only know the type. Remember, that is the power of free theorems: being able to derive statements about functions without knowing their defining equations.

$\underline{\alpha \to \alpha}$ The first example is the free theorem for the CuMin type $\alpha \to \alpha$, which demonstrates the approach well, despite the derived statement itself being rather obvious (since there are not many functions of that type). Let be given CuMin types τ_1 and τ_2 (both closed), a function symbol f with type annotation $f :: \forall\alpha.\, \alpha \to \alpha$ in the given program, and terms g and x with $\vdash g :: \tau_1 \to \tau_2$ and $\vdash x :: \tau_1$. We would like to prove the CuMin equivalence $g\ (f_{\tau_1}\ x) \equiv f_{\tau_2}\ (g\ x)$. Here α is ε-quantified, so it can be instantiated with any type and the anything-primitive cannot be used for this type.

The claim can be stated (via translation and one of the laws) as the following SaLT equivalence:

$$\lceil g \rceil \ni g' \cup \lceil f_{\tau_1} \rceil \ni f' \cup \lceil x \rceil \ni x' \cup f'\ x' \ni y \cup g'\ y$$
$$\equiv \lceil g \rceil \ni g' \cup \lceil f_{\tau_2} \rceil \ni f' \cup \lceil x \rceil \ni x' \cup g'\ x' \ni z \cup f'\ z$$

Even though the statement (just) happens to be true in general for strict g, for the derivation we assume g to be also multi-deterministic; i.e., we have

$$\lceil g \rceil \equiv sMap\ (\lambda\, \hat{g}' :: \lfloor\tau_1\rfloor \to \lfloor\tau_2\rfloor.\, \lambda\, x :: \lfloor\tau_1\rfloor.\, \{\hat{g}'\ x\})\ \hat{g}$$
$$\equiv \hat{g} \ni \hat{g}' \cup \{\{\_\} \circ \hat{g}'\}$$

for some $\vdash \hat{g} :: \{\lfloor\tau_1\rfloor \to \lfloor\tau_2\rfloor\}$. Using this to replace $\lceil g \rceil$ in the above, then applying (5), and then getting rid of the variable g' by replacing it with $\{\_\} \circ \hat{g}'$ according to (3), leads to:

$$\hat{g} \ni \hat{g}' \cup \lceil f_{\tau_1} \rceil \ni f' \cup \lceil x \rceil \ni x' \cup f'\ x' \ni y \cup (\{\_\} \circ \hat{g}')\ y$$
$$\equiv \hat{g} \ni \hat{g}' \cup \lceil f_{\tau_2} \rceil \ni f' \cup \lceil x \rceil \ni x' \cup (\{\_\} \circ \hat{g}')\ x' \ni z \cup f'\ z$$

which after some further manipulations becomes:

$$\hat{g} \ni \hat{g}' \cup f^t_{\lfloor\tau_1\rfloor} \ni f' \cup \lceil x \rceil \ni x' \cup sMap\ \hat{g}'\ (f'\ (id\ x'))$$
$$\equiv \hat{g} \ni \hat{g}' \cup f^t_{\lfloor\tau_2\rfloor} \ni f' \cup \lceil x \rceil \ni x' \cup sMap\ id\ (f'\ (\hat{g}'\ x'))$$

Now, since f is a function symbol with type annotation $f :: \forall\alpha.\, \alpha \to \alpha$ in the original program, its translation is a function symbol with type annotation $f^t :: \forall\alpha.\, \{\alpha \to \{\alpha\}\}$ in the translated program, so the following typing judgment is valid:

$$\alpha^\varepsilon, in :: \lfloor\tau_1\rfloor \to \alpha, out :: \alpha \to \lfloor\tau_2\rfloor$$
$$\vdash f^t_\alpha \ni f' \cup \lceil x \rceil \ni x' \cup sMap\ out\ (f'\ (in\ x')) :: \{\lfloor\tau_2\rfloor\}$$

Hence, Theorem 8.3 can be used with the term from that typing judgment as e', and indeed closes the gap. Strictness of g is really necessary in this example, since the given function could have been $f\ x = \mathsf{failure}_\alpha$.

$\underline{\alpha \to \alpha \to (\alpha, \alpha)}$ In CuMin, if $f :: \forall\alpha.\, \alpha \to \alpha \to (\alpha, \alpha)$ is a function symbol, τ_1 and τ_2 closed types, $g :: \tau_1 \to \tau_2$ a strict and multi-deterministic function, and $x, y :: \tau_1$ terms, then we can show

$$pMap\ g\ (f_{\tau_1}\ x\ y) \equiv \mathsf{let}\ g' = g\ \mathsf{in}\ f_{\tau_2}\ (g'\ x)\ (g'\ y)$$

where *pMap* is the function given at the beginning of Section 2. This time, the statement is more interesting than in the previous example, since there are many more possible functions of the given type, e.g.,

$f\ x\ y = (x, x \cup y) \cup (y, \mathsf{failure}_\alpha)$. The proof is similar to the one of the previous example: translate all expressions to SaLT, make use of g being multi-deterministic, and then apply Theorem 8.3 to the following expression:

$$\hat{g} \ni \hat{g}' \bigcup f'_\alpha \ni f' \bigcup \lceil x \rceil \ni x' \bigcup \lceil y \rceil \ni y' \bigcup f'\ (in\ x') \ni c \bigcup \\ c\ (in\ y') \ni b \bigcup \mathsf{case}\ b\ \mathsf{of}\ \langle (u,v) \to \{(out\ u, out\ v)\} \rangle$$

Here we really need g multi-deterministic, as $f\ x\ y = (x,x)$ and $g\ x = x \cup (x+1)$ would constitute a counterexample otherwise. Note also that on the right-hand side of the overall claim, we use a let-binding to share a common value of g throughout the expression, which is what allows us to deal with multi-deterministic g. This actually is the normal case: Theorem 8.3 always produces two semantically equivalent expressions starting with an indexed union $\hat{g} \ni \hat{g}' \bigcup \ldots$, which fits the $\mathsf{let}\ g' = g\ \mathsf{in}$ in the CuMin equivalence. However, the left-hand side here uses g only once, and within the function *pMap* sharing is enforced by call-time choice. Thus, for the left-hand side (as well as for both sides of the previous example) the question of sharing is irrelevant and "$\mathsf{let}\ g' = g\ \mathsf{in}$" can be dropped.

$\underline{\forall^* \alpha.(\alpha, \alpha)}$ In CuMin, if $c :: \forall^* \alpha.(\alpha, \alpha)$ is a polymorphic top-level constant, τ_1 and τ_2 are closed data types, and $g :: \tau_1 \to \tau_2$ is a strict, multi-deterministic, and multi-onto function, then

$$pMap\ g\ c_{\tau_1} \ \equiv \ c_{\tau_2}$$

where *pMap* is given as before. This time we have to require g to be multi-onto, because the quantification over α is now restricted to data types and thus c can generate values of type α using the anything primitive. For example, the equivalence does not hold for $c = (\mathsf{anything}_\alpha, \mathsf{anything}_\alpha)$ and the strict, multi-deterministic, but not multi-onto $g = h \cup k$ of type $\mathsf{Nat} \to \mathsf{Bool}$, where $h\ x = (x \equiv x)$ and $k\ x = (x \equiv (x+1))$, despite the fact that $g\ \mathsf{anything}_{\mathsf{Nat}} \equiv \mathsf{anything}_{\mathsf{Bool}}$ holds. (Hint: $pMap\ g\ c_{\mathsf{Nat}}$ lacks the pairs (**False**, **True**) and (**True**, **False**).)

$\underline{[\alpha] \to [\alpha]}$ In order to deal with lists in CuMin, we need the *map* function with type $map :: \forall\alpha.\forall\beta.(\alpha \to \beta) \to [\alpha] \to [\beta]$ which applies the first argument to every entry in the second argument. Using this function, we can state the free theorem for functions $f :: \forall\alpha.[\alpha] \to [\alpha]$: For closed types τ_1 and τ_2, a strict and multi-deterministic function $g :: \tau_1 \to \tau_2$, and a term $x :: [\tau_1]$, we can show that $map\ g\ (f_{\tau_1}\ x) \equiv f_{\tau_2}\ (map\ g\ x)$, i.e., the example from the introduction (the interesting bit being what conditions on g do guarantee the validity – and now we do know for sure).

About the proof: In SaLT we define a function $mapM :: \forall\alpha.\forall\beta.(\alpha \to \{\beta\}) \to [\alpha] \to \{[\beta]\}$ that applies a set-valued function to every entry of a list independently and combines the possible results for each entry to overall generate a set of lists. This function is named after the Haskell function *mapM* and has similar properties, for example $mapM_{\tau\tau}\ \{\_\}_\tau \equiv \{\_\}_{[\tau]}$ as can be checked easily. One can also check that $\lceil map\ g \rceil \equiv \lceil g \rceil \ni g' \bigcup \{mapM\ g'\}$, which allows to easily translate applications of CuMin *map* into SaLT. With these insights we can prove the intended CuMin free theorem.

10. Conclusion

We have shown how to derive free theorems for functional-logic programming languages like Curry. The utility of free theorems for functional languages has been demonstrated by a variety of uses in the past and we hope the results presented here will help to yield functional-logic counterparts. Our results depend on side conditions constraining nondeterminism, and further investigations will have to show how restrictive these conditions actually are. It is plausible to assume that there are other side conditions under which similar free theorems can be shown, e.g., restrictions concerning sharing rather than nondeterminism. While a lot remains to be done, this paper shows that there are results to be found and that they can be proved in a formally rigorous way.

At the same time, this paper introduces a new denotational semantics for a sublanguage of Curry, which can serve as a basis for further research also apart from parametricity and free theorems. It has helped to promote our understanding of Curry with regards to the interaction of laziness, recursion, and the different flavors of nondeterminism, and we hope others will profit as well.

Acknowledgments

We thank all the reviewers involved in the development of this paper for their criticism and advice.

References

[1] E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. Operational semantics for declarative multi-paradigm languages. *J. Symb. Comput.*, 40(1):795–829, 2005.

[2] B. Braßel and F. Huch. On a tighter integration of functional and logic programming. In *APLAS, Proceedings*, volume 4807 of *LNCS*, pages 122–138. Springer, 2007.

[3] B. Braßel, S. Fischer, M. Hanus, and F. Reck. Transforming functional logic programs into monadic functional programs. In *WFLP 2010, Revised Selected Papers*, volume 6559 of *LNCS*, pages 30–47. Springer, 2011.

[4] J. Christiansen, D. Seidel, and J. Voigtländer. Free theorems for functional logic programs. In *PLPV, Proceedings*, pages 39–48. ACM, 2010.

[5] J. Christiansen, D. Seidel, and J. Voigtländer. An adequate, denotational, functional-style semantics for Typed FlatCurry. In *WFLP 2010, Revised Selected Papers*, volume 6559 of *LNCS*, pages 119–136. Springer, 2011.

[6] J. Christiansen, D. Seidel, and J. Voigtländer. An adequate, denotational, functional-style semantics for Typed FlatCurry without Letrec. Technical report, University of Bonn, 2011. `http://www.iai.uni-bonn.de/~jv/IAI-TR-2011-1.pdf`.

[7] J. Christiansen, M. Hanus, F. Reck, and D. Seidel. A semantics for weakly encapsulated search in functional logic programs. In *PPDP, Proceedings*, pages 49–60. ACM, 2013.

[8] A. Gill, J. Launchbury, and S. Peyton Jones. A short cut to deforestation. In *FPCA, Proceedings*, pages 223–232. ACM, 1993.

[9] J. González-Moreno, M. Hortalá-González, F. López-Fraguas, and M. Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *J. Log. Program.*, 40(1):47–87, 1999.

[10] M. Hanus. Functional logic programming: From theory to Curry. In *Programming Logics — Essays in Memory of Harald Ganzinger*, volume 7797 of *LNCS*, pages 123–168. Springer, 2013.

[11] H. Hußmann. Nondeterministic algebraic specifications and nonconfluent term rewriting. *J. Log. Program.*, 12(3&4):237–255, 1992.

[12] F. López-Fraguas and J. Sánchez-Hernández. *TOY*: A multiparadigm declarative system. In *RTA, Proceedings*, volume 1631 of *LNCS*, pages 244–247. Springer, 1999.

[13] F. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. Equivalence of two formal semantics for functional logic programs. In *PROLE 2006, Proceedings*, volume 188 of *ENTCS*, pages 117–142. Elsevier, 2007.

[14] R. Møgelberg and A. Simpson. Relational parametricity for computational effects. *Log. Meth. Comput. Sci.*, 5(3), 2009.

[15] J. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing, Proceedings*, pages 513–523. Elsevier, 1983.

[16] J. Voigtländer. Bidirectionalization for free! In *POPL, Proceedings*, pages 165–176. ACM, 2009.

[17] P. Wadler. Theorems for free! In *FPCA, Proceedings*, pages 347–359. ACM, 1989.