Institut für Informatik III

der Rheinischen Friedrich-Wilhelms-Universität Bonn

An Adequate, Denotational, Functional-Style Semantics for Typed FlatCurry without Letrec

Jan Christiansen, Daniel Seidel,
and Janis Voigtländer

IAI-TR-2011-1 März 2011

INSTITUT FÜR INFORMATIK III

DER RHEINISCHEN FRIEDRICH-WILHELMS-UNIVERSITÄT BONN

An Adequate, Denotational, Functional-Style Semantics for Typed FlatCurry without Letrec

Jan Christiansen, Daniel Seidel,
and Janis Voigtländer

IAI-TR-2011-1 März 2011

Forschungsbericht
Technical Report

ISSN 0944-8535

An Adequate, Denotational, Functional-Style Semantics for Typed FlatCurry without Letrec[*]

Jan Christiansen[1], Daniel Seidel[2†], and Janis Voigtländer[2]

[1]Christian-Albrechts-Universität Kiel, Institut für Informatik,
Olshausenstraße 40, 24098 Kiel, Germany.
E-mail: `jac@informatik.uni-kiel.de`

[2]Rheinische Friedrich-Wilhelms-Universität Bonn, Institut für Informatik,
Römerstraße 164, 53117 Bonn, Germany.
E-mail: `{ds,jv}@informatik.uni-bonn.de`

Abstract

With the aim of putting type-based reasoning for functional logic languages, as recently explored by Christiansen et al. (2010), on a formal basis, we develop a denotational semantics for a typed core language of Curry. Dealing with the core language FlatCurry rather than with full Curry suffices, since there exists a type-preserving translation from the latter into the former. In contrast to existing semantics for functional logic languages, we deliberately approach the problem "from the functional side". That is, rather than adapting approaches previously known from the study of (resolution-like) semantics for logic languages, we aim for a semantics in the spirit of standard denotational semantics for the polymorphic lambda calculus. We claim and set out to prove that the presented semantics is adequate with respect to an existing operational semantics. Due to problems with recursive let-bindings in combination with call-time choice (discussed in an appendix), we give the denotational semantics in the presence of non-recursive let-expressions only.

1 Introduction

It would be nice to have standard reasoning tools used in functional languages at disposal for functional logic languages as well. For example, type-based reasoning via free theorems (Wadler 1989), which has become popular in the functional programming community, might also be useful for functional logic languages. Free theorems formalize semantic properties that are obtained by considering only the type of a function. We have already investigated free theorems for the functional logic language Curry in an informal and example-driven way (Christiansen et al. 2010), but a thorough formal investigation is still missing. To formally investigate free theorems, we have to define a logical relation over the type structure of the programming language and to prove fundamental results about it. For doing so, a completely compositional semantics, a denotational semantics, is very desirable. Here we mean compositionality less in the sense of module structure and related concepts for programming-in-the-large as studied by Molina-Bravo and Pimentel (2003), and more in the basic sense considered also by López-Fraguas et al. (2009).

In the setting of functional logic programming, there are two nearly standard semantics: the constructor based rewrite logic CRWL of González-Moreno et al. (1999) and an

[*]This paper is a revised version of a WFLP'10 contribution (Christiansen et al. 2011).

[†]This author was supported by the DFG under grant VO 1512/1-1.

operational semantics of Albert et al. (2005) and Braßel and Huch (2007a). In the absence of recursive let-expressions, López-Fraguas et al. (2007) have proved these semantics equivalent. The rewriting logic CRWL consists of a set of (conditional) rewrite rules, and programs are rewritten to a set of constructor terms. Originally, CRWL does not consider types and does not support higher-order functions. Even if known extensions for these aspects are considered, a denotational semantics seems more valuable for our purpose. Hence, the first step in a formal investigation of free theorems for Curry is the development of an appropriate denotational semantics for at least a subset of the language, containing the main functional logic features. Equational reasoning is another beneficiary. Finally, a functional-style semantics fits in well with recent developments in the field of functional logic programming. For example, while early Curry implementations compiled to Prolog, recent implementations compile to Haskell.

A functional logic language can be considered as a functional language extended with nondeterminism and free variables. Adding nondeterminism can be modeled by switching from a single-value term semantics to a set-value term semantics. Søndergaard and Sestoft (1992) present twelve possible choices in the design space for this kind of denotational semantics. These choices span over three independent issues, namely strict or non-strict evaluation, call-time or run-time choice, and angelic, demonic, or erratic nondeterminism. Curry is a non-strict language with call-time choice and essentially provides angelic nondeterminism. Call-time choice means that for each variable a nondeterministic choice (if necessary) is made only once, even if the variable is used more than once. It is maybe best explained by the `double coin` example, where `coin` is nondeterministically `0` or `1` and `double` is defined as `double x = x + x`. Since the choice for `coin` to be either `0` or `1` is the same for both occurrences of `x` in the body of `double`, under call-time choice the result of `double coin` is `0` or `2` but never `1`.

Now, what should a denotational, functional-style semantics for functional logic programs be like? Of course, it should be equivalent to an established semantics. Moreover, choosing a "native" source language seems reasonable. Here we present a denotational, functional-style semantics for the typed, flat, functional logic language TFLC. This language is a typed adaptation of the FLC language, which is the source language for the operational semantics of Albert et al. (2005) and for a CRWL-adaptation by López-Fraguas et al. (2007), who introduced the name. The language reflects all semantically important features of FlatCurry, the intermediate language for all recent Curry implementations (MCC, PAKCS, KiCS), but does not deal with implementation features like modules and import statements. Also, for our denotational semantics we allow only non-recursive let-expressions. Since there exists a type-preserving translation from Curry to FlatCurry and hence to TFLC, we are hopeful that type-based reasoning about TFLC programs will eventually allow assertions about actual Curry programs.

The contributions of this paper are the extension of the FLC language with a type system and the definition of a completely compositional, denotational, functional-style semantics for the typed language TFLC minus recursive let-expressions. We also start to give formal evidence that our semantics then is equivalent to the operational semantics of Albert et al. (2005) and Braßel and Huch (2007a), and therefore, equivalent as well to CRWL (López-Fraguas et al. 2007). In an appendix, we consider the situation with recursive let-expressions.

2 The Language TFLC

First, we give the syntax of TFLC:

$$
\begin{array}{lcl}
\sigma & ::= & \forall\alpha.\sigma \mid \tau \\
\tau & ::= & \alpha \mid \tau_1 \rightarrow \tau_2 \mid [\tau] \mid \mathsf{Nat} \mid \mathsf{Bool} \\
P & ::= & D\ P \mid \epsilon \\
D & ::= & f :: \sigma;\ f(\overline{x_n}) = e \\
e & ::= & x \mid n \mid e_1 + e_2 \mid \mathsf{Nil}_\tau \mid \mathsf{Cons}(e_1, e_2) \mid \mathsf{True} \mid \mathsf{False} \mid e_1\ ?\ e_2 \mid \\
 & & \textbf{case}\ e\ \textbf{of}\ \ \{\mathsf{True} \rightarrow e_1; \mathsf{False} \rightarrow e_2\} \mid f_{\overline{\tau_n}} \mid \textbf{apply}(e_1, e_2) \mid \textbf{unknown}_\tau \mid \\
 & & \textbf{case}\ e\ \textbf{of}\ \ \{\mathsf{Nil} \rightarrow e_1; \mathsf{Cons}(x, xs) \rightarrow e_2\} \mid \textbf{let}\ \overline{x_n :: \tau_n = e_n}\ \textbf{in}\ e \mid \textbf{failed}_\tau
\end{array}
$$

In it, σ denotes a type scheme (possibly with leading type variable quantifications), τ a type (without type variable quantifications), P a program, D a function definition with type declaration, and e an expression. Moreover, α ranges over type variables, x over term variables, n over natural numbers, and ϵ represents the empty program. We refer to $\mathsf{Cons}, \mathsf{Nil}_\tau, \mathsf{True}, \mathsf{False}$, and any natural number n as *constructor symbols* and to f with $(f(\overline{x_n}) = e) \in P$ as (n-ary) *function symbol*. An expression whose outermost symbol is a constructor symbol is called *constructor-rooted*. We use bars to refer to sequences of objects. For example, $\overline{x_n :: \tau_n = e_n}$ denotes $x_1 :: \tau_1 = e_1, \ldots, x_n :: \tau_n = e_n$.

Our TFLC syntax differs in some respects from the syntax of FLC as presented by Albert et al. (2005). We add $\textbf{unknown}_\tau$ to denote free variables and $\textbf{failed}_\tau$ to denote failure. Albert et al. instead use recursive let-expressions of the form $\textbf{let}\ x = x\ \textbf{in}\ \ldots$ to denote free variables, and introduce failure only implicitly via incomplete case expressions. These are only minor differences. Higher-order is addressed through the primitive $\textbf{apply}(\cdot,\cdot)$ and a syntactic form for type-instantiated function symbols, which is similar to the extension made by Albert et al. (2005) in their Section 5.3. Also compared to their FLC language, we omit rigid case expressions, because we do not consider residuation. Moreover, for simplicity, we consider only lists and Booleans as algebraic data types. However, the denotational semantics to be presented in Section 4 can easily be extended to arbitrary algebraic data types. As a primitive data type, we include natural numbers, called Nat. While current implementations like PAKCS suspend if a free variable of type Int is instantiated, our data type Nat is narrowable.

Two more comments on the treatment of free variables are in order. 1) Any occurrence of $\textbf{unknown}_\tau$ denotes a fresh occurrence of a free variable, not a single constant. Thus, the expressions $\textbf{let}\ x :: \mathsf{Nat} = \textbf{unknown}_{\mathsf{Nat}}\ \textbf{in}\ x + x$ and $\textbf{let}\ x :: \mathsf{Nat} = \textbf{unknown}_{\mathsf{Nat}}, y :: \mathsf{Nat} = \textbf{unknown}_{\mathsf{Nat}}\ \textbf{in}\ x+y$ mean very different things. 2) Free variables of function types (generally, of types involving $\rightarrow$) are not permitted. We could have encoded this constraint in the type system, presented next, but for simplicity leave it as a global restriction instead.

Figure 1 shows the typing rules for TFLC. A typing judgment is of the form $\Gamma \vdash e :: \tau$ and states that e is typable to τ under an (unordered) typing context Γ. Such a typing judgment, with typing context $\Gamma = \{\overline{\alpha_m}, \overline{x_n :: \tau_n}\}$, is *valid* if and only if all type variables occurring in $\overline{\tau_n}$ and τ are among $\overline{\alpha_m}$, all term variables occurring unbound in e are among $\overline{x_n}$, and there exists a type derivation for the judgment using the rules given in Figure 1. Note that typing 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, the typing judgment thus would need to be indexed by that program P, but we omit this for the sake of readability. By $[\tau/\alpha]$ we mean syntactic replacement of occurrences of a type variable α by a type τ, and

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

$$\Gamma, x :: \tau \vdash x :: \tau \qquad \Gamma \vdash \mathbf{unknown}_\tau :: \tau \qquad \Gamma \vdash \mathbf{failed}_\tau :: \tau$$

$$\frac{\Gamma \vdash e_1 :: \tau \qquad \Gamma \vdash e_2 :: [\tau]}{\Gamma \vdash \mathsf{Cons}(e_1, e_2) :: [\tau]} \qquad \frac{\Gamma \vdash e :: \mathsf{Bool} \qquad \Gamma \vdash e_1 :: \tau \qquad \Gamma \vdash e_2 :: \tau}{\Gamma \vdash \mathbf{case}\ e\ \mathbf{of}\ \{\mathsf{True} \to e_1; \mathsf{False} \to e_2\} :: \tau}$$

$$\frac{\Gamma \vdash e :: [\tau'] \qquad \Gamma \vdash e_1 :: \tau \qquad \Gamma, x :: \tau', xs :: [\tau'] \vdash e_2 :: \tau}{\Gamma \vdash \mathbf{case}\ e\ \mathbf{of}\ \{\mathsf{Nil} \to e_1; \mathsf{Cons}(x, xs) \to e_2\} :: \tau}$$

$$\frac{\Gamma \vdash e_1 :: \mathsf{Nat} \quad \Gamma \vdash e_2 :: \mathsf{Nat}}{\Gamma \vdash e_1 + e_2 :: \mathsf{Nat}} \qquad \frac{\Gamma \vdash e_1 :: \tau \quad \Gamma \vdash e_2 :: \tau}{\Gamma \vdash e_1\ ?\ e_2 :: \tau}$$

$$\frac{\Gamma, \overline{x_n :: \tau_n} \vdash e_1 :: \tau_1 \quad \cdots \quad \Gamma, \overline{x_n :: \tau_n} \vdash e_n :: \tau_n \qquad \Gamma, \overline{x_n :: \tau_n} \vdash e :: \tau}{\Gamma \vdash \mathbf{let}\ \overline{x_n :: \tau_n = e_n}\ \mathbf{in}\ e :: \tau}$$

$$\frac{(f :: \forall\alpha_1 \cdots \forall\alpha_m.\tau) \in P}{\Gamma \vdash f_{\overline{\tau_m}} :: \tau[\overline{\tau_m/\alpha_m}]} \qquad \frac{\Gamma \vdash e_1 :: \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 :: \tau_1}{\Gamma \vdash \mathbf{apply}(e_1, e_2) :: \tau_2}$$

Figure 1: Typing rules for TFLC

we use a corresponding vector notation for multiple simultaneous replacements. A program P is well-typed if and only if for each function definition $f(\overline{x_n}) = e$ with type declaration $f :: \forall\alpha_1.\cdots\forall\alpha_m.\tau_1 \to \cdots \to \tau_n \to \tau$ in P the typing judgment $\overline{\alpha_m}, \overline{x_n :: \tau_n} \vdash e :: \tau$ is valid.

3 The Natural Semantics

Albert et al. (2005) present an operational semantics, called natural semantics, for FLC. To guarantee correct behavior with respect to call-time choice, the natural semantics is defined for a normalized subset of FLC terms and programs. We adapt this natural semantics to TFLC. Therefore, we need to restrict to a normalized subset of TFLC expressions and programs as well.

Definition 3.1. A TFLC expression is *normalized* if

- all arguments to **apply** are variables,
- all constructor arguments are variables,
- $\mathbf{unknown}_\tau$ only appears as right-hand side of definitions in let-expressions.

A TFLC program P is *normalized* if for every function definition $f(\overline{x_n}) = e$ in P the right-hand side e is normalized.

The process of normalization introduces let-expressions that bind (non-variable) function arguments to variables. The original function arguments are replaced by these variables. For a formal specification of normalization we refer to Albert et al. (2005, Definition 3.2). An adaptation of this procedure to TFLC is possible but cumbersome, due to the required handling of type annotations. We avoid spelling out the details, and simply let $e^\dagger$ and $P^\dagger$ denote the normalized versions of a term e and a program P, respectively.

Figure 2 presents the natural semantics for normalized TFLC. Note that we consider a fixed normalized program P, which, for example, is used in rule ($\textsc{Fun}$). We omit rules for Boolean case expressions as they are analogous to the rules for list case expressions. We deal with higher-order features in the spirit of the corresponding extension of Albert et al. (2005, Section 5.4.3). Note that we do not take over the problematic rule ($\textsc{VarExp}$) of Albert et al. (2005), instead replacing it by the rule ($\textsc{Lookup}$) of Braßel and Huch (2007a, who discovered the inappropriateness of ($\textsc{VarExp}$)).

Given a normalized expression e under a *heap* Δ, a semantic derivation for a statement $\Delta : e \Downarrow \Delta' : v$ means that e under that heap can be evaluated to v while the heap changes to Δ'. The derivation guarantees that v is in *head normal form* or a variable that points to $\mathbf{unknown}_\tau$ in Δ'. An expression is in head normal form if it is constructor-rooted or a partial function application, i.e., of the form $f_{\overline{\tau_m}}(\overline{e_k})$, which is an abbreviation for $\mathbf{apply}(\dots(\mathbf{apply}(f_{\overline{\tau_m}}, e_1), \dots), e_k)$, for some n-ary function symbol f and $k < n$. Note that partial applications with $k > 1$ are never part of a normalized TFLC expression; but they may arise in semantic derivations in the premise of the rule ($\textsc{Apply}$). Also, for $k = 0$ we identify $f_{\overline{\tau_m}}(\overline{e_k}) = f_{\overline{\tau_m}}()$ with just $f_{\overline{\tau_m}}$.

For details about the rules of the natural semantics we refer to the papers of Albert et al. (2005) and Braßel and Huch (2007a). But let us highlight some adaptations that are due to our syntactic handling of free variables and failure. The rule ($\textsc{Lookup}$) forces the heap entry for the considered variable to be evaluated if it is not some $\mathbf{unknown}_\tau$. If it *is* $\mathbf{unknown}_\tau$, the rule ($\textsc{Val}$) applies instead, and the evaluation is completed. Thus, a free variable is only instantiated if necessary, that is, if it appears in the scrutinee of a case expression or in a summation. For example, in the rule ($\textsc{LGuess}_2$) the free variable x of type $[\tau]$ is instantiated to $\mathsf{Cons}(y, ys)$ to go on with the derivation via the second premise of the rule. The function ϱ used, for example, in ($\textsc{LGuess}_2$) renames variables and is canonically extended to arbitrary expressions. By employing (globally) fresh variables in ($\textsc{LGuess}_2$) and in ($\textsc{Let}$), we guarantee that no variable is ever used twice.

As an example, we present a derivation for the `double coin` example used in the introduction. We consider the following TFLC program:

$$\begin{gathered} \mathit{coin} :: \mathsf{Nat};\ \mathit{coin}() = 0 \mathbin{?} 1 \\ \mathit{double} :: \mathsf{Nat} \to \mathsf{Nat};\ \mathit{double}(x) = x + x \end{gathered}$$

Fortunately, both functions are already in normalized form. On the other hand, the expression $\mathbf{apply}(\mathit{double}, \mathit{coin})$, i.e., `double coin` in TFLC syntax, is not. So we consider the corresponding normalized version $\mathbf{let}\ x = \mathit{double}, y = \mathit{coin}\ \mathbf{in}\ \mathbf{apply}(x, y)$.[1] In the following derivation we use the abbreviations $\Delta_1 = \{x' \mapsto \mathit{double}, y' \mapsto \mathit{coin}\}$, $\Delta_2 = \{x' \mapsto \mathit{double}\}$, and $\Delta_3 = \{x' \mapsto \mathit{double}, y' \mapsto 0\}$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\{y' \mapsto \mathit{coin}\} : \mathit{double} \Downarrow \{y' \mapsto \mathit{coin}\} : \mathit{double}}\,(\textsc{PartVal})}{\Delta_1 : x' \Downarrow \Delta_1 : \mathit{double}}\,(\textsc{Lookup}) \qquad (1)}{\Delta_1 : \mathbf{apply}(x', y') \Downarrow \Delta_3 : 0}\,(\textsc{Apply})}{\emptyset : \mathbf{let}\ x = \mathit{double}, y = \mathit{coin}\ \mathbf{in}\ \mathbf{apply}(x, y) \Downarrow \Delta_3 : 0}\,(\textsc{Let})$$

[1]Here, and later, we omit type annotations for let-bound variables in examples.

(Lookup)	$\dfrac{\Delta : e \Downarrow \Delta' : v}{\Delta[x \mapsto e] : x \Downarrow \Delta'[x \mapsto v] : v}$	where $e \neq \textbf{unknown}_\tau$ for any τ
(Val)	$\Delta : v \Downarrow \Delta : v$	where v is constructor-rooted or $v = x$ and $\Delta(x) = \textbf{unknown}_\tau$ for some τ
(Fun)	$\dfrac{\Delta : \varrho(e) \Downarrow \Delta' : v}{\Delta : f_{\overline{\tau_m}}(\overline{x_n}) \Downarrow \Delta' : v}$	where $f :: \forall\alpha_1.\cdots\forall\alpha_m.\tau;\ f(\overline{y_n}) = e$ in P and $\varrho = \{\overline{\alpha_m \mapsto \tau_m}, \overline{y_n \mapsto x_n}\}$
(Let)	$\dfrac{\Delta[\overline{y_k \mapsto \varrho(e_k)}] : \varrho(e) \Downarrow \Delta' : v}{\Delta : \textbf{let}\ \overline{x_k :: \tau_k = e_k}\ \textbf{in}\ e \Downarrow \Delta' : v}$	where $\varrho = \{\overline{x_k \mapsto y_k}\}$ with $\overline{y_k}$ fresh
(Plus$_1$)	$\dfrac{\Delta : e_1 \Downarrow \Delta' : n_1 \qquad \Delta' : e_2 \Downarrow \Delta'' : n_2}{\Delta : e_1 + e_2 \Downarrow \Delta'' : n_1 + n_2}$	
(Plus$_2$)	$\dfrac{\Delta : e_1 \Downarrow \Delta' : n_1 \qquad \Delta' : e_2 \Downarrow \Delta''[y \mapsto \textbf{unknown}_{\mathsf{Nat}}] : y}{\Delta : e_1 + e_2 \Downarrow \Delta''[y \mapsto n_2] : n_1 + n_2}$	
(Plus$_3$)	$\dfrac{\Delta : e_1 \Downarrow \Delta'[x \mapsto \textbf{unknown}_{\mathsf{Nat}}] : x \qquad \Delta'[x \mapsto n_1] : e_2 \Downarrow \Delta'' : n_2}{\Delta : e_1 + e_2 \Downarrow \Delta'' : n_1 + n_2}$	
(Plus$_4$)	$\dfrac{\begin{array}{c}\Delta : e_1 \Downarrow \Delta'[x \mapsto \textbf{unknown}_{\mathsf{Nat}}] : x \\ \Delta'[x \mapsto n_1] : e_2 \Downarrow \Delta''[y \mapsto \textbf{unknown}_{\mathsf{Nat}}] : y\end{array}}{\Delta : e_1 + e_2 \Downarrow \Delta''[y \mapsto n_2] : n_1 + n_2}$	where $n_1, n_2 \in \mathsf{Nat}$
(Or$_i$)	$\dfrac{\Delta : e_i \Downarrow \Delta' : v}{\Delta : e_1\ ?\ e_2 \Downarrow \Delta' : v}$	where $i \in \{1, 2\}$
(LSelect$_1$)	$\dfrac{\Delta : e \Downarrow \Delta' : \mathsf{Nil}_\tau \qquad \Delta' : e_1 \Downarrow \Delta'' : v}{\Delta : \textbf{case}\ e\ \textbf{of}\ \{\mathsf{Nil} \mapsto e_1; \mathsf{Cons}(x_1, x_2) \mapsto e_2\} \Downarrow \Delta'' : v}$	
(LSelect$_2$)	$\dfrac{\Delta : e \Downarrow \Delta' : \mathsf{Cons}(y, ys) \qquad \Delta' : \varrho(e_2) \Downarrow \Delta'' : v}{\Delta : \textbf{case}\ e\ \textbf{of}\ \{\mathsf{Nil} \mapsto e_1; \mathsf{Cons}(x_1, x_2) \mapsto e_2\} \Downarrow \Delta'' : v}$	where $\varrho = \{x_1 \mapsto y, x_2 \mapsto ys\}$
(LGuess$_1$)	$\dfrac{\Delta : e \Downarrow \Delta'[x \mapsto \textbf{unknown}_{[\tau]}] : x \qquad \Delta'[x \mapsto \mathsf{Nil}_\tau] : e_1 \Downarrow \Delta'' : v}{\Delta : \textbf{case}\ e\ \textbf{of}\ \{\mathsf{Nil} \mapsto e_1; \mathsf{Cons}(x_1, x_2) \mapsto e_2\} \Downarrow \Delta'' : v}$	
(LGuess$_2$)	$\dfrac{\Delta : e \Downarrow \Delta'[x \mapsto \textbf{unknown}_{[\tau]}] : x \qquad \Delta' \cup \Delta'' : \varrho(e_2) \Downarrow \Delta''' : v}{\Delta : \textbf{case}\ e\ \textbf{of}\ \{\mathsf{Nil} \mapsto e_1; \mathsf{Cons}(x_1, x_2) \mapsto e_2\} \Downarrow \Delta''' : v}$	where $\varrho = \{x_1 \mapsto y, x_2 \mapsto ys\}$ with y, ys fresh and $\Delta'' = [x \mapsto \mathsf{Cons}(y, ys), y \mapsto \textbf{unknown}_\tau, ys \mapsto \textbf{unknown}_{[\tau]}]$
(PartVal)	$\Delta : f_{\overline{\tau_m}}(\overline{x_k}) \Downarrow \Delta : f_{\overline{\tau_m}}(\overline{x_k})$	
(Apply)	$\dfrac{\Delta : x \Downarrow \Delta' : f_{\overline{\tau_m}}(\overline{x}_k) \qquad \Delta' : f_{\overline{\tau_m}}(\overline{x_k}, y) \Downarrow \Delta'' : v}{\Delta : \textbf{apply}(x, y) \Downarrow \Delta'' : v}$	where $f :: \forall\alpha_1.\ldots\forall\alpha_m.\tau;\ f(\overline{x_n}) = e$ in P and $k < n$

Figure 2: Adaptation of the natural semantics presented by Braßel and Huch (2007a)

At position (1) we use the following subderivation:[2]

$$
\begin{prooftree}
\AxiomC{}
\RightLabel{(\textsc{Val})}
\UnaryInfC{$\Delta_2 : 0 \Downarrow \Delta_2 : 0$}
\RightLabel{(\textsc{Or}$_1$)}
\UnaryInfC{$\Delta_2 : 0 \mathbin{?} 1 \Downarrow \Delta_2 : 0$}
\RightLabel{(\textsc{Fun})}
\UnaryInfC{$\Delta_2 : \mathit{coin} \Downarrow \Delta_2 : 0$}
\RightLabel{(\textsc{Lookup})}
\UnaryInfC{$\Delta_1 : y' \Downarrow \Delta_3 : 0$}
\AxiomC{}
\RightLabel{(\textsc{Val})}
\UnaryInfC{$\Delta_2 : 0 \Downarrow \Delta_2 : 0$}
\RightLabel{(\textsc{Lookup})}
\UnaryInfC{$\Delta_3 : y' \Downarrow \Delta_3 : 0$}
\RightLabel{(\textsc{Plus}$_1$)}
\BinaryInfC{$\Delta_1 : y' + y' \Downarrow \Delta_3 : 0$}
\RightLabel{(\textsc{Fun})}
\UnaryInfC{$\Delta_1 : \mathit{double}(y') \Downarrow \Delta_3 : 0$}
\end{prooftree}
$$

Of course, another derivation is possible as well: instead of ($\textsc{Or}_1$) we can use ($\textsc{Or}_2$) and then derive the final result 2. This highlights that nondeterminism is handled by competing rules in the natural semantics. To get the overall possible semantics of a single expression, each possible derivation has to be considered. The same holds for CRWL.

Given the example calculation, we observe pros and cons of using an operational semantics. On the one hand, the natural semantics is very useful because it is very close to a Curry implementation. On the other hand, for calculations by hand and for formal reasoning the semantics is not that advantageous. The lengthy derivations and in particular the contribution of potentially many derivations to the semantics of a single expression are hindering. Another drawback is the need for normalized expressions.

A denotational semantics as given in the next section is very much complementary to an operational semantics. It is more abstract and thereby, as a con, farther from an implementation. On the pro side, every calculation is closed in itself. As calculations thus become more concise, it is easier to calculate by hand. Moreover, the semantics is completely compositional, meaning that the semantics of each subterm of a term can be computed separately and then be used directly to calculate the semantics of the whole term. This makes that semantics suitable in particular for equational reasoning, and for proofs of semantic properties by induction on the structure of expressions. Other advantages are that it operates on all (not only normalized) well-typed TFLC terms, and that sharing effects are very explicit, which makes them easier to understand.

4 The Denotational Semantics

As we want to model nondeterminism, we cannot use single-value functions as semantic objects. Walicki and Meldal (1997) give an overview over appropriate function models. The main criterion for choosing a function model is the decision about call-time vs. run-time choice. As Curry uses call-time choice, the appropriate function models are 1) the *poweralgebraic* model with the restriction to *additive* functions and 2) the *multialgebraic* model.

The poweralgebraic model interprets each function as a mapping that maps *a set to a set*. This model can be used for both strategies, run-time and call-time choice. If we consider only additive functions, we get a model for call-time choice. A (semantic) function f is additive if for all input sets X, Y we have $f(X \cup Y) = f(X) \cup f(Y)$. That is, (at least on finite sets) every function is completely determined by its behavior on the empty set and on all singleton sets. To satisfy additivity within a term semantics, we have to ensure that

[2]Note that $\mathit{double}(y')$ is, by the convention introduced two pages earlier, an abbreviation for $\mathbf{apply}(\mathit{double}, y')$.

shared variables embody the same nondeterministic choice. In terms of the `double coin` example, we have to guarantee that the semantics of `double coin` is the union of the semantics of `double 0` and `double 1`. There are basically two approaches to guarantee additivity of functions in the term semantics. We can define function application by passing an input set, to a function, piecewise as singleton sets (and maybe only pass through the empty set, with some extra care to guarantee laziness). Or we can carefully delay sharing as much as possible, by having an eye on the function input whenever it is used more than once in the function body. The latter approach is preferable in an implementation as presented by Braßel et al. (2011). But it is rather unwieldy in the design of a denotational semantics, because we would have to either deeply investigate the syntax of terms or be overly conservative by suspecting a necessity for sharing whenever evaluation of a term splits into the evaluation of several subterms. Thus, the first alternative mentioned above seems more appropriate, that is, pass an input set, to a function, piecewise as singleton sets.

Nevertheless, we choose still another model, namely a multialgebraic one, as it turns out that the mentioned kind of poweralgebraic model is isomorphic to the multialgebraic model. A function that takes only singleton sets or the empty set can also be considered as a function that takes single elements, where the empty set is modeled as an additional special element. Mapping *an element to a set* (instead of *a set to a set*) corresponds to the multialgebraic approach. We choose the multialgebraic instead of the poweralgebraic model as, in our opinion, it provides a better overall intuition about the actual behavior of functions. Also, the multialgebraic model is quite similar to the CRWL approach, and thus likely to be more accessible to readers familiar with existing semantics for functional logic languages.

Another choice to be made in a nondeterministic setting is the treatment of failure. One of the arguments of a nondeterministic choice can be a failure or even be nonterminating. So far we have not considered whether a nondeterministic choice is supposed to return all non-failure results, only a part of them, or maybe no result at all in this case. These three possible approaches are called angelic, erratic, and demonic nondeterminism, respectively. Curry provides (as far as reasonable[3]) angelic nondeterminism, returning all non-failure results. Thus, for example, the terms 3 and 3 ? $\mathbf{failed}_{\mathsf{Nat}}$ are assigned the same semantics.

The appropriate way to model angelic nondeterminism is the Hoare powerdomain. If we restrict ourselves to directed-complete partial orders (dcpos) as domains, an adaptation of Theorem 6.2.13 of Abramsky and Jung (1994) allows us to define the Hoare powerdomain in terms of Scott-closed subsets. A subset A of a dcpo $(D, \sqsubseteq)$ is *Scott-closed* if it is a lower set that is closed under the suprema of directed subsets. A subset A of a dcpo $(D, \sqsubseteq)$ is a *lower set* if $x \in A$ implies that all $y \in D$ with $y \sqsubseteq x$ are in A, too. A *directed subset* A is a non-empty subset where every two elements in A have a supremum in A.

Definition 4.1. Let $\mathbf{D} = (D, \sqsubseteq)$ be a dcpo. Its *Hoare powerdomain* $\mathcal{P}_H(\mathbf{D})$ is the complete lattice of all Scott-closed subsets of D, along with order $\subseteq$ on those.

Infimum and supremum of $M \subseteq \mathcal{P}_H(\mathbf{D})$ are defined by:

$$\bigcap M = \{x \in D \mid \forall m \in M.\ x \in m\}$$
$$\bigsqcup M = \bigcap \{m \in \mathcal{P}_H(\mathbf{D}) \mid \forall n \in M.\ n \subseteq m\}.$$

[3]No implementation is angelic w.r.t. nonterminating computations. Even if all non-failure results are yielded, such a computation will still fail to terminate afterwards.

With the just given definition of the Hoare powerdomain we have fixed the domain structure for interpreting types to dcpos. The type semantics is defined by:

$$\begin{aligned}
[\![\alpha]\!]_\theta &= \theta(\alpha) \\
[\![\mathsf{Bool}]\!]_\theta &= \{\mathit{True}, \mathit{False}\} \\
[\![\mathsf{Nat}]\!]_\theta &= \mathbb{N} \\
[\![\tau_1 \to \tau_2]\!]_\theta &= \{f : ([\![\tau_1]\!]_\theta)_\perp \to \mathcal{P}_H([\![\tau_2]\!]_\theta) \mid f \text{ continuous}\} \backslash \{\Omega\} \\
[\![[\tau]]\!]_\theta &= \mathit{lfp}(\lambda S.\{[\,]\} \cup \{a : b \mid a \in ([\![\tau]\!]_\theta)_\perp, b \in S_\perp\})
\end{aligned}$$

where θ is a given mapping from type variables to dcpos, to provide for polymorphic types. Some explanations are in order here, given next.

Types are interpreted as the domains of individual semantic objects, not as the corresponding powerdomains. Therefore, the semantics of a term of type τ will be in the powerdomain of τ's semantics. The semantics of Bool and Nat are sets with the discrete order, in particular *without* least elements. Results summarized by Abramsky and Jung (1994) guarantee that the domain structure of type semantics is preserved by the constructions for function and for list types. Regarding the function space, functions map *one element* of the input type to *a set of elements* of the output type, i.e., are mappings from a domain to a powerdomain. Failure (essentially the empty set) as input to a function needs extra handling, as mentioned before. It is not present in any type's domain, but is conceptually a valid input to a function, hence we add it as a special element $\perp$. The lifting operator $(\cdot)_\perp$ adds such an element as least element to a dcpo. Continuity (Scott-continuity, i.e., monotonicity and preservation of suprema of directed sets) of functions is enforced explicitly to guarantee that the function space itself is a dcpo (Abramsky and Jung 1994). The order on the function space is point-wise, and the *least defined function* $\Omega = \lambda a.\emptyset$ is excluded from the function space, as we identify it with failure (which, as indicated above, is not represented in the domains and instead comes in as the empty set in the respective powerdomain). The semantics of list types is given via least fixpoints (denoted via *lfp*). The entries in a list are single elements, not sets. All nondeterminism in a list is "flattened out" and is represented by having a set of deterministic lists. When we consider constructors as functions, this approach corresponds to the multialgebraic function model. This also motivates having $\perp$ as a possible list element. The order on the interpretation of list types is by element-wise comparison.

As mentioned earlier, for the denotational treatment we restrict TFLC to use only non-recursive let-expressions. We then define the term semantics w.r.t. a fixed program P, as presented in Figure 3. There, θ maps each type variable to a dcpo and σ maps each term variable to a single element from a pointed dcpo, specifically *not* to an element from a powerdomain over such a dcpo. The latter (restriction to single elements) guarantees call-time choice. For example, in the semantics of $[\![f_{\overline{\tau_m}}]\!]$ the extended variable binding $\sigma[\overline{x_n \mapsto \mathbf{a_n}}]$ stores possible choices for actual function arguments individually, rather than a set of possible values for each formal parameter x_i.

To guarantee that all term semantics are Scott-closed sets, i.e., elements of a powerdomain, we employ the operation $(\cdot)\!\downarrow$, called down-closure. In a dcpo $(D, \sqsubseteq)$, we define $A\!\downarrow = \{x \in D \mid \exists y \in A.\ x \sqsubseteq y\}$ for every $A \subseteq D$. We say that $A\!\downarrow$ is the lower set generated by A. Down-closure suffices for our purpose to guarantee Scott-closedness since we use it only on finite sets. The following two lemmas establish that fact.

$$\llbracket x \rrbracket_{\theta,\sigma} = \begin{cases} \emptyset & \text{if } \sigma(x) = \bot \\ \{\sigma(x)\}{\downarrow} & \text{otherwise} \end{cases} \qquad \llbracket n \rrbracket_{\theta,\sigma} = \{n\}$$

$$\llbracket \mathsf{True} \rrbracket_{\theta,\sigma} = \{\mathit{True}\} \qquad \llbracket \mathsf{False} \rrbracket_{\theta,\sigma} = \{\mathit{False}\} \qquad \llbracket \mathsf{Nil}_\tau \rrbracket_{\theta,\sigma} = \{[\,]\}$$

$$\llbracket \mathsf{Cons}(e_1, e_2) \rrbracket_{\theta,\sigma} = \bigsqcup_{\mathbf{h} \in (\llbracket e_1 \rrbracket_{\theta,\sigma})_\bot} \bigsqcup_{\mathbf{t} \in (\llbracket e_2 \rrbracket_{\theta,\sigma})_\bot} \{\mathbf{h} : \mathbf{t}\}$$

$$\llbracket e_1 + e_2 \rrbracket_{\theta,\sigma} = \bigsqcup_{\mathbf{a} \in \llbracket e_1 \rrbracket_{\theta,\sigma}} \bigsqcup_{\mathbf{b} \in \llbracket e_2 \rrbracket_{\theta,\sigma}} \{\mathbf{a} + \mathbf{b}\} \qquad \llbracket e_1 \mathbin{?} e_2 \rrbracket_{\theta,\sigma} = \llbracket e_1 \rrbracket_{\theta,\sigma} \cup \llbracket e_2 \rrbracket_{\theta,\sigma}$$

$$\llbracket \mathbf{unknown}_\tau \rrbracket_{\theta,\sigma} = \llbracket \tau \rrbracket_\theta \qquad \llbracket \mathbf{failed}_\tau \rrbracket_{\theta,\sigma} = \emptyset$$

$$\llbracket f_{\overline{\tau_m}} \rrbracket_{\theta,\sigma} = (\{\lambda \mathbf{a_1}. \ldots (\{\lambda \mathbf{a_n}. \llbracket e \rrbracket_{\theta[\overline{\alpha_m \mapsto \llbracket \tau_m \rrbracket_\theta}], \sigma[\overline{x_n \mapsto \mathbf{a_n}}]}\} \setminus \{\Omega\}){\downarrow} \ldots\} \setminus \{\Omega\}){\downarrow}$$
$$\text{with } f :: \forall \alpha_1. \cdots \forall \alpha_m. \tau;\ f(\overline{x_n}) = e \text{ in } P$$

$$\llbracket \mathbf{apply}(e_1, e_2) \rrbracket_{\theta,\sigma} = \bigsqcup_{\mathbf{f} \in \llbracket e_1 \rrbracket_{\theta,\sigma}} \bigsqcup_{\mathbf{a} \in (\llbracket e_2 \rrbracket_{\theta,\sigma})_\bot} (\mathbf{f}\ \mathbf{a})$$

$$\llbracket \mathbf{case}\ e\ \mathbf{of}\ \{\mathsf{Nil} \to e_1; \mathsf{Cons}(x_1, x_2) \to e_2\} \rrbracket_{\theta,\sigma} =$$
$$\bigsqcup_{\mathbf{t} \in \llbracket e \rrbracket_{\theta,\sigma}} \begin{cases} \llbracket e_1 \rrbracket_{\theta,\sigma} & \text{if } \mathbf{t} = [\,] \\ \llbracket e_2 \rrbracket_{\theta,\sigma[x_1 \mapsto \mathbf{t}_1, x_2 \mapsto \mathbf{t}_2]} & \text{if } \mathbf{t} = \mathbf{t}_1 : \mathbf{t}_2 \end{cases}$$

$$\llbracket \mathbf{case}\ e\ \mathbf{of}\ \{\mathsf{True} \to e_1; \mathsf{False} \to e_2\} \rrbracket_{\theta,\sigma} = \bigsqcup_{\mathbf{t} \in \llbracket e \rrbracket_{\theta,\sigma}} \begin{cases} \llbracket e_1 \rrbracket_{\theta,\sigma} & \text{if } \mathbf{t} = \mathit{True} \\ \llbracket e_2 \rrbracket_{\theta,\sigma} & \text{if } \mathbf{t} = \mathit{False} \end{cases}$$

$$\llbracket \mathbf{let}\ \overline{x_n :: \tau_n = e_n}\ \mathbf{in}\ e \rrbracket_{\theta,\sigma} = \bigsqcup_{(\overline{\mathbf{t}_n}) \in \mathbf{T}_{\overline{x_n = e_n}}} \llbracket e \rrbracket_{\theta,\sigma[\overline{x_n \mapsto \mathbf{t}_n}]}$$
$$\text{with } \mathbf{T}_{\overline{x_n = e_n}} \text{ the set of all tuples } (\overline{\mathbf{t}_n}) \text{ that fulfill the following requirements:}$$
$$\mathbf{t}_1 \in (\llbracket e_1 \rrbracket_{\theta,\sigma[\overline{x_n \mapsto \mathbf{t}_n}]})_\bot, \ldots, \mathbf{t}_n \in (\llbracket e_n \rrbracket_{\theta,\sigma[\overline{x_n \mapsto \mathbf{t}_n}]})_\bot$$

Figure 3: Denotational term semantics

Lemma 4.2. *Let $(D, \sqsubseteq)$ be a dcpo and $A \subseteq D$ such that all directed subsets of A have an upper bound in A. Then $A{\downarrow}$ is Scott-closed.*

Lemma 4.3. *In every finite partially ordered set A each directed subset has an upper bound in A.*

In the term semantics for a variable we use a down-closure w.r.t. the dcpo that is the semantics of the type of x. Moreover, we use down-closure in the term semantics for function symbols. Note that there we have to remove Ω before each application of down-closure because the type semantics for function types do not contain Ω.

Since the system of equations in Figure 3 is recursive, in general there can be more than one solution. We take the canonical (least) one.

Definition 4.4. The semantic function $\llbracket\cdot\rrbracket_{\cdot,\cdot}$ w.r.t. a program P, formally is a family of functions $\llbracket\cdot\rrbracket^\tau_{\cdot,\cdot}$, indexed by the type τ (subsequently suppressed in notation), where each $\llbracket\cdot\rrbracket^\tau_{\cdot,\cdot}$ maps terms e with $\Gamma \vdash e :: \tau$, and corresponding environments θ, σ, to elements of

$\mathcal{P}_H(\llbracket\tau\rrbracket_\theta)$. Here, "corresponding" means that θ maps every type variable in Γ to a dcpo, while σ maps every term variable $x :: \tau'$ in Γ to an element of $(\llbracket\tau'\rrbracket_\theta)_\perp$. We take $\llbracket\cdot\rrbracket_{\cdot,\cdot}$ as the least solution of the recursive system of equations in Figure 3 ("least" in the point-wise order on functions into $\mathcal{P}_H(\llbracket\tau\rrbracket_\theta)$ for τ, θ, σ).

Instead of Scott-closed sets, it often suffices to calculate with sets that contain the same maximal elements as a Scott-closed set. Additionally, most $\bigsqcup$-operators can simply be considered set unions, as stated by the following lemmas.

Lemma 4.5. *The semantic function $\llbracket\cdot\rrbracket_{\cdot,\cdot}$ is monotone w.r.t. the term environment entries. That is, $\llbracket e\rrbracket_{\theta,\sigma[x\mapsto\mathbf{t}]} \subseteq \llbracket e\rrbracket_{\theta,\sigma[x\mapsto\mathbf{t}']}$ whenever $\mathbf{t} \sqsubseteq \mathbf{t}'$ in the appropriate domain.*

Lemma 4.6. *Let $(D, \sqsubseteq)$ be a dcpo and A be a Scott-closed set in D. Then $A = A'{\downarrow}$ for every $A' \subseteq A$ with the same maximal elements as A.*

Lemma 4.7. *Let $\mathbf{D} = (D, \sqsubseteq)$, $\mathbf{E} = (E, \sqsubseteq')$ be dcpos, and $A \subseteq D$. Let f be a monotone map from $\mathbf{D}_\perp$ to $\mathcal{P}_H(\mathbf{E})$. We have $\bigsqcup_{a\in A\downarrow}(f\ a) = \bigsqcup_{a\in A}(f\ a)$, and if A is finite, then $\bigsqcup_{a\in A}(f\ a) = \bigcup_{a\in A}(f\ a)$.*

This simplifies calculating with the denotational semantics, as shown in the following example for **apply**(*double*, *coin*):

$$\llbracket coin\rrbracket_{\emptyset,\emptyset} = \llbracket 0\ ?\ 1\rrbracket_{\emptyset,\emptyset} = \llbracket 0\rrbracket_{\emptyset,\emptyset} \cup \llbracket 1\rrbracket_{\emptyset,\emptyset} = \{0,1\}$$

$$\begin{aligned}
\llbracket double\rrbracket_{\emptyset,\emptyset} &= (\{\lambda\mathbf{a}.\llbracket x+x\rrbracket_{\emptyset,[x\mapsto\mathbf{a}]}\} \setminus \{\Omega\}){\downarrow}\\
&= (\{\lambda\mathbf{a}.\textstyle\bigsqcup_{\mathbf{b}\in\llbracket x\rrbracket_{\emptyset,[x\mapsto\mathbf{a}]}}\bigsqcup_{\mathbf{c}\in\llbracket x\rrbracket_{\emptyset,[x\mapsto\mathbf{a}]}}\{\mathbf{b}+\mathbf{c}\}\} \setminus \{\Omega\}){\downarrow}\\
&= (\{\lambda\mathbf{a}.\textstyle\bigsqcup_{\mathbf{b}\in\llbracket x\rrbracket_{\theta,[x\mapsto\mathbf{a}]}}\begin{cases}\bigsqcup_{\mathbf{c}\in\emptyset}\{\mathbf{b}+\mathbf{c}\} & \text{if } \mathbf{a}=\perp\\ \bigsqcup_{\mathbf{c}\in\{\mathbf{a}\}\downarrow}\{\mathbf{b}+\mathbf{c}\} & \text{otherwise}\end{cases}\} \setminus \{\Omega\}){\downarrow}\\
&= (\{\lambda\mathbf{a}.\textstyle\bigsqcup_{\mathbf{b}\in\llbracket x\rrbracket_{\theta,[x\mapsto\mathbf{a}]}}\begin{cases}\bigsqcup_{\mathbf{c}\in\emptyset}\{\mathbf{b}+\mathbf{c}\} & \text{if } \mathbf{a}=\perp\\ \bigsqcup_{\mathbf{c}\in\{\mathbf{a}\}}\{\mathbf{b}+\mathbf{c}\} & \text{otherwise}\end{cases}\} \setminus \{\Omega\}){\downarrow}\\
&= (\{\lambda\mathbf{a}.\textstyle\bigsqcup_{\mathbf{b}\in\llbracket x\rrbracket_{\theta,[x\mapsto\mathbf{a}]}}\begin{cases}\bigcup_{\mathbf{c}\in\emptyset}\{\mathbf{b}+\mathbf{c}\} & \text{if } \mathbf{a}=\perp\\ \bigcup_{\mathbf{c}\in\{\mathbf{a}\}}\{\mathbf{b}+\mathbf{c}\} & \text{otherwise}\end{cases}\} \setminus \{\Omega\}){\downarrow}\\
&= (\{\lambda\mathbf{a}.\textstyle\bigsqcup_{\mathbf{b}\in\llbracket x\rrbracket_{\theta,[x\mapsto\mathbf{a}]}}\begin{cases}\emptyset & \text{if } \mathbf{a}=\perp\\ \{\mathbf{b}+\mathbf{a}\} & \text{otherwise}\end{cases}\} \setminus \{\Omega\}){\downarrow}\\
&= (\{\lambda\mathbf{a}.\begin{cases}\emptyset & \text{if } \mathbf{a}=\perp\\ \{\mathbf{a}+\mathbf{a}\} & \text{otherwise}\end{cases}\} \setminus \{\Omega\}){\downarrow}\\
&= (\{\lambda\mathbf{a}.\begin{cases}\emptyset & \text{if } \mathbf{a}=\perp\\ \{\mathbf{a}+\mathbf{a}\} & \text{otherwise}\end{cases}\}){\downarrow}
\end{aligned}$$

$$\begin{aligned}
\llbracket\mathbf{apply}(double, coin)\rrbracket_{\emptyset,\emptyset} &= \textstyle\bigsqcup_{\mathbf{f}\in\llbracket double\rrbracket_{\emptyset,\emptyset}}\bigsqcup_{\mathbf{a}\in(\llbracket coin\rrbracket_{\emptyset,\emptyset})_\perp}(\mathbf{f}\ \mathbf{a})\\
&= \bigsqcup_{\mathbf{f}\in(\{\lambda\mathbf{a}.\begin{cases}\emptyset & \text{if } \mathbf{a}=\perp\\ \{\mathbf{a}+\mathbf{a}\} & \text{otherwise}\end{cases}\})\downarrow}\textstyle\bigsqcup_{\mathbf{a}\in(\llbracket coin\rrbracket_{\emptyset,\emptyset})_\perp}(\mathbf{f}\ \mathbf{a})
\end{aligned}$$

$$
\begin{aligned}
&= \bigcup_{\mathbf{f} \in \{\lambda \mathbf{a}. \begin{cases} \emptyset & \text{if } \mathbf{a} = \bot \\ \{\mathbf{a}+\mathbf{a}\} & \text{otherwise} \end{cases}\}} \bigsqcup_{\mathbf{a} \in (\llbracket coin \rrbracket_{\emptyset,\emptyset})_\bot} (\mathbf{f}\ \mathbf{a}) \\
&= \bigsqcup_{\mathbf{a} \in \{0,1\}_\bot} \begin{cases} \emptyset & \text{if } \mathbf{a} = \bot \\ \{\mathbf{a}+\mathbf{a}\} & \text{otherwise} \end{cases} \\
&= \{0, 2\}
\end{aligned}
$$

As a more complex example, consider a program containing

$$ones :: [\mathsf{Nat}];\ \ ones() = \mathsf{Nil}_{\mathsf{Nat}}\ ?\ \mathsf{Cons}(1, ones)$$

and then the following calculation:

$$
\begin{aligned}
\llbracket ones \rrbracket_{\emptyset,\emptyset} &= \llbracket \mathsf{Nil}_{\mathsf{Nat}}\ ?\ \mathsf{Cons}(1, ones) \rrbracket_{\emptyset,\emptyset} \\
&= \llbracket \mathsf{Nil}_{\mathsf{Nat}} \rrbracket_{\emptyset,\emptyset} \cup \llbracket \mathsf{Cons}(1, ones) \rrbracket_{\emptyset,\emptyset} \\
&= \{[\,]\} \cup \bigsqcup_{\mathbf{t} \in (\llbracket ones \rrbracket_{\emptyset,\emptyset})_\bot} \{\bot : \mathbf{t}, 1 : \mathbf{t}\} \\
&= \{[\,]\} \cup \bigsqcup_{\mathbf{t} \in (\{[\,]\} \cup \bigsqcup_{\mathbf{t}' \in (\llbracket ones \rrbracket_{\emptyset,\emptyset})_\bot} \{\bot : \mathbf{t}', 1 : \mathbf{t}'\})_\bot} \{\bot : \mathbf{t}, 1 : \mathbf{t}\} \\
&= \{[\,], \bot : \bot, 1 : \bot, \bot : [\,], 1 : [\,]\} \cup \bigsqcup_{\mathbf{t} \in \bigsqcup_{\mathbf{t}' \in (\llbracket ones \rrbracket_{\emptyset,\emptyset})_\bot} \{\bot : \mathbf{t}', 1 : \mathbf{t}'\}} \{\bot : \mathbf{t}, 1 : \mathbf{t}\} \\
&= \{[\,], \bot : \bot, 1 : \bot, \bot : [\,], 1 : [\,], \bot : \bot : \bot, 1 : \bot : \bot, \bot : 1 : \bot, 1 : 1 : \bot, \bot : \bot : [\,], \\
&\qquad 1 : \bot : [\,], \bot : 1 : [\,], 1 : 1 : [\,], \ldots\} \\
&= \{[\,], 1 : [\,], 1 : 1 : [\,], \ldots, 1 : 1 : \cdots\}{\downarrow}
\end{aligned}
$$

5 Equivalence of the Semantics

As a goal in designing a denotational semantics for Curry, the new semantics should be equivalent to an already established one. We claim that, in the absence of recursive let-expressions, the denotational semantics from Section 4 is equivalent to the natural semantics in Section 3. There are several notions of equivalence of two semantics. Obviously, there is no chance for a statement like "the same expressions evaluate to the same semantic values" here since the natural semantics yields a set of terms while the denotational semantics yields a set of mathematical objects. Instead, we relate the denotational semantics of a "result" of a derivation in the natural semantics with the denotational semantics of the original expression. This corresponds to the well-established notion of computational adequacy of a denotational semantics with respect to an operational semantics. Since the natural semantics requires normalized expressions, we first restrict ourselves to normalized expressions and programs in what follows. At the end of this section we extend the results to general TFLC expressions and programs (clearly, still without recursive let-expressions).

For an equivalence statement we have to relate the environments under which the natural and the denotational semantics operate. We use the fact that we are interested only in well-typed expressions, and hence relate the environments via a typing context. Note that during the proofs we employ implicitly that the natural semantics preserves well-typedness at each step of a derivation. By $dom(\cdot)$ we denote the domain of a function. Since we consider only non-recursive let-expressions, we can restrict attention to non-recursive heaps, i.e., heaps where no entry directly or indirectly depends on itself.

Definition 5.1. Let $\Gamma = \{\overline{\alpha_k}, \overline{x_l :: \tau_l}\}$ be a typing context. We say (θ, Δ, σ) corresponds to Γ, written $(\theta, \Delta, \sigma) \sim \Gamma$, if Δ is a non-recursive heap, $dom(\theta) = \{\overline{\alpha_k}\}$, $dom(\Delta) \cup dom(\sigma) = \{\overline{x_l}\}$, $dom(\Delta) \cap dom(\sigma) = \emptyset$, $\forall x \in dom(\Delta).\ \Gamma \vdash \Delta(x) :: \tau$, and $\forall x \in dom(\sigma).\ \sigma(x) \in (\llbracket \tau \rrbracket_\theta)_\perp$, where $\Gamma \vdash x :: \tau$.

The relation $\sim$ extends the notion "corresponding" from Definition 4.4 and allows us to define the denotational semantics of a heap-expression pair.

Definition 5.2. Let $\Gamma \vdash e :: \tau$ and $(\theta, \Delta, \sigma) \sim \Gamma$ with $\Delta = \{\overline{x_n \mapsto e_n}\}$. We set

$$\llbracket \Delta, e \rrbracket_{\theta,\sigma} = \llbracket \textbf{let } \overline{x_n :: \tau_n = e_n} \textbf{ in } e \rrbracket_{\theta,\sigma}$$

where the $\overline{\tau_n}$ are taken from the $\overline{x_n :: \tau_n}$ entries in Γ, which are guaranteed to exist by the correspondence relation.

Now we are able to state the main conjecture, establishing equivalence of the natural and the denotational semantics.

Conjecture 5.3. *If $\Gamma \vdash e :: \tau$, $(\theta, \Delta, \sigma) \sim \Gamma$, and $\forall x \in dom(\sigma).\ \sigma(x) = \perp$, then, in the absence of recursive let-expressions,*

$$\llbracket \Delta, e \rrbracket_{\theta,\sigma} = \textstyle\bigsqcup_{\Delta:e \Downarrow \Delta':v} \llbracket \Delta', v \rrbracket_{\theta,\sigma}.$$

Note that the right-hand side still contains a nontrivial invocation of the denotational semantics, because the operational semantics only ever reduces to a head normal form. That the above adequacy statement is indeed very useful for relating denotational and operational semantics can be seen by looking at its specialization to ground types. For example, if we set τ to Nat, we obtain that $\llbracket \Delta, e \rrbracket_{\theta,\sigma}$ contains an $n \in \mathbb{N}$ if and only if there exist Δ' and v with $\Delta : e \Downarrow \Delta' : v$ and either $v = n$ or v is a variable x with $\Delta'(x) = \textbf{unknown}_{\mathsf{Nat}}$.

We split Conjecture 5.3 into the following theorem and conjecture, where each can be used to establish one inclusion of the desired equivalence.

Theorem 5.4. *Let $\Gamma \vdash e :: \tau$. Then, in the absence of recursive let-expressions,*

$$\Delta : e \Downarrow \Delta' : v \quad \Rightarrow \quad \llbracket \Delta, e \rrbracket_{\theta,\sigma} \supseteq \llbracket \Delta', v \rrbracket_{\theta,\sigma}$$

whenever $(\theta, \Delta, \sigma) \sim \Gamma$ (without further condition on σ!).

That is, $\llbracket \Delta, e \rrbracket_{\theta,\sigma}$ is an upper bound for each $\llbracket \Delta', v \rrbracket_{\theta,\sigma}$ with $\Delta : e \Downarrow \Delta' : v$, so clearly it is above or equal to the *least* upper bound $\bigsqcup_{\Delta:e \Downarrow \Delta':v} \llbracket \Delta', v \rrbracket_{\theta,\sigma}$, which gives one inclusion direction of Conjecture 5.3.

For the other direction, we need some preparation. By a standard fixpoint theorem (using that $\llbracket \cdot \rrbracket_{\theta,\sigma}$ is continuous), we can characterize the semantic function from Definition 4.4 via a "step-indexed" version. We set $\llbracket e \rrbracket^0_{\theta,\sigma} = \emptyset$. For every $i > 0$ we get $\llbracket e \rrbracket^i_{\theta,\sigma}$ via an adaptation of the rules in Figure 3 by replacing all occurrences of some $\llbracket e' \rrbracket_{\theta',\sigma'}$ on the right-hand sides by $\llbracket e' \rrbracket^{i-1}_{\theta',\sigma'}$. For example, if $e = e_1 \ ?\ e_2$, then for every $i > 0$ the semantics $\llbracket e \rrbracket^i_{\theta,\sigma}$ is defined as $\llbracket e_1 \rrbracket^{i-1}_{\theta,\sigma} \cup \llbracket e_2 \rrbracket^{i-1}_{\theta,\sigma}$. Then it holds in general that $\llbracket e \rrbracket_{\theta,\sigma} = \bigsqcup_{i \in \mathbb{N}} \llbracket e \rrbracket^i_{\theta,\sigma}$. Via Definition 5.2, the step-index is transferred to the notion $\llbracket \Delta, e \rrbracket_{\theta,\sigma}$ as well, and we can state the following conjecture.

Conjecture 5.5. *Let $\Gamma \vdash e :: \tau$, $(\theta, \Delta, \sigma) \sim \Gamma$, and $\forall x \in dom(\sigma).\ \sigma(x) = \bot$. Then, in the absence of recursive let-expressions, for every $i \in \mathbb{N}$ and $\mathbf{t} \in [\![\Delta, e]\!]^i_{\theta,\sigma}$ there exists a natural semantics derivation yielding $\Delta : e \Downarrow \Delta' : v$ such that $\mathbf{t} \in [\![\Delta', v]\!]_{\theta,\sigma}$.*

This gives the inclusion of $[\![\Delta, e]\!]_{\theta,\sigma}$ in $\bigsqcup_{\Delta:e\Downarrow\Delta':v}[\![\Delta', v]\!]_{\theta,\sigma}$ from Conjecture 5.3, because: $[\![\Delta, e]\!]_{\theta,\sigma}$ is $\bigsqcup_{i\in\mathbb{N}}[\![\Delta, e]\!]^i_{\theta,\sigma}$, Conjecture 5.5 gives that $\bigsqcup_{\Delta:e\Downarrow\Delta':v}[\![\Delta', v]\!]_{\theta,\sigma}$ is an upper bound for each $[\![\Delta, e]\!]^i_{\theta,\sigma}$, and hence it is above or equal to the *least* upper bound $\bigsqcup_{i\in\mathbb{N}}[\![\Delta, e]\!]^i_{\theta,\sigma}$.

We have proved Theorem 5.4, by induction on the length of the semantic derivation for $\Delta : e \Downarrow \Delta' : v$.[4] The proof uses several lemmas and is structured similarly to a proof of López-Fraguas et al. (2007, proof of Theorem 6.2). Here, we only provide the main results contributing to the proof.

First, we show how the concept of context refinement is represented in the denotational semantics.

Lemma 5.6. *Let $\Gamma \vdash e :: \tau$, $\Gamma' \vdash e :: \tau$, $(\theta, \Delta, \sigma) \sim \Gamma$, and $(\theta, \Delta', \sigma) \sim \Gamma'$. Then, in the absence of recursive let-expressions,*

$$(\forall x \in dom(\Delta).\ [\![\Delta, x]\!]_{\theta,\sigma} \supseteq [\![\Delta', x]\!]_{\theta,\sigma}) \quad \Rightarrow \quad [\![\Delta, e]\!]_{\theta,\sigma} \supseteq [\![\Delta', e]\!]_{\theta,\sigma}$$

Next, we split the effects of a natural semantics derivation into effects on the heap and on the result. (The proof uses Lemma 5.6.)

Lemma 5.7. *Let $\Gamma \vdash e :: \tau$ and $(\theta, \Delta, \sigma) \sim \Gamma$. If $\Delta : e \Downarrow \Delta' : v$, then, in the absence of recursive let-expressions:*

1. *$\forall x \in dom(\Delta).\ [\![\Delta, x]\!]_{\theta,\sigma} \supseteq [\![\Delta', x]\!]_{\theta,\sigma}$, and*
2. *$[\![\Delta', e]\!]_{\theta,\sigma} \supseteq [\![\Delta', v]\!]_{\theta,\sigma}$.*

Combining Lemma 5.6 and Lemma 5.7(1), we get a corollary that is the last ingredient necessary for the proof of Theorem 5.4.

Corollary 5.8. *Let $\Gamma \vdash e :: \tau$ and $(\theta, \Delta, \sigma) \sim \Gamma$. If $\Delta : e \Downarrow \Delta' : v$, then, in the absence of recursive let-expressions, $[\![\Delta, e]\!]_{\theta,\sigma} \supseteq [\![\Delta', e]\!]_{\theta,\sigma}$.*

Proof (Theorem 5.4). Let $\Gamma \vdash e :: \tau$ and let (θ, Δ, σ) be corresponding to Γ. Assume $\Delta : e \Downarrow \Delta' : v$. Then by Corollary 5.8 we have $[\![\Delta, e]\!]_{\theta,\sigma} \supseteq [\![\Delta', e]\!]_{\theta,\sigma}$, and by Lemma 5.7(2) we have $[\![\Delta', e]\!]_{\theta,\sigma} \supseteq [\![\Delta', v]\!]_{\theta,\sigma}$. Hence, $[\![\Delta, e]\!]_{\theta,\sigma} \supseteq [\![\Delta', v]\!]_{\theta,\sigma}$.

Finally, in the absence of recursive let-expressions, denotational results about normalized TFLC terms can be extended to all TFLC terms by the following lemma.

Lemma 5.9. *Let P be a program and e be a well-typed expression such that $\emptyset \vdash e :: \tau$. It holds that, in the absence of recursive let-expressions,*

$$[\![e]\!]^P_{\emptyset,\emptyset} = [\![e^\dagger]\!]^{P^\dagger}_{\emptyset,\emptyset}\,,$$

where the superscripts indicate the programs that are used to calculate the semantics.

[4]For Conjecture 5.5 we expect that the stratification via i will facilitate a proof. That is, a proof would be first by induction on $i \in \mathbb{N}$, then by structural induction on e.

6 Conclusion

We have presented a new semantics for a core of the functional logic language Curry. It is the first functional-style, denotational semantics for this language. We partially proved equivalence to an existing natural semantics (Albert et al. 2005; Braßel and Huch 2007a). We expect benefits for equational reasoning and for formally establishing type-based reasoning principles. Problems arising with recursive let-expressions are discussed in Appendix A.

Acknowledgments

We would like to thank Rudolf Berghammer and Achim Jung for discussions about (power)domain theoretic issues, and David Sabel for pointing us in the right direction regarding problems with denotational semantics for recursive let.

A What about Recursive Let-Expressions?

We have avoided treating recursive let-expressions in our denotational semantics here. That is a pity, because we think that the combination of call-time choice and recursive let is under-appreciated. For example, we can employ a recursive let-expression to define an interesting way of swapping two elements in a list. This example is inspired by a similar example of Erkök (2002), who uses the list monad in Haskell instead of nondeterminism. We define a function called `replace`, which takes an element and a list and nondeterministically replaces one entry in the list by that element. The function yields the resulting list and the list entry that has been replaced.

```
replace :: Int -> [Int] -> ([Int],Int)
replace x (y:ys) =
  (x:ys,y) ? let (zs,z) = replace x ys in (y:zs,z)
```

By means of `replace`, we now define a function `pairSwaps` that nondeterministically exchanges two elements in a list:

```
pairSwaps :: [Int] -> [Int]
pairSwaps xs = let (ys,y) = replace z xs
                   (zs,z) = replace y ys in zs
```

While being useful, even correctly implementing call-time choice in the presence of let-recursion seems problematic, as exhibited below. First, we give the following Curry definition of `take`, which for an input list `l` yields the longest prefix of `l` with at most `n` ≥ 0 elements:

```
take :: Int -> [a] -> [a]
take n l = if n<=0 then [] else take' n l
  where
    take' _ []     = []
    take' m (x:xs) = x : take (m-1) xs
```

Now, consider the following Curry definition, where `?` denotes nondeterministic choice:

```
ones = [] ? (1 : ones)
```

When we use KiCSi (the interpreter of the Kiel Curry System KiCS) and ask for the list with the first three (if existing) elements of `ones`, we get the following answers:

```
> take 3 ones
[]
More?

[1]
More?

[1,1]
More?

[1,1,1]
More?

No more Solutions
```

But if we consider a very similar definition, just moving `ones` into a let-expression, KiCSi behaves differently:

```
> let ones = [] ? (1:ones) in take 3 ones
[]
More?

[1,1,1]
More?

No more Solutions
```

Because of call-time choice, the decision whether `ones` is `[]` or `1:ones` is made just once for the let-bound `ones`. For the top-level `ones` before, in contrast, the choice is made in every single recursive step. From a term-rewriting point of view, in the first definition `ones` is a *constant*, i.e., a nullary function, while in the second definition `ones` is a *variable*. That is, in the top-level definition `ones` is simply a name for its right-hand side while in the let-bound definition `ones` is a variable that obeys call-time choice. Omitting `take 3` in both examples, the first definition thus yields lists of `1`s of every length while the second definition yields only the empty list and the infinite list of `1`s.

So far, so good. But the semantics of recursive let-expressions appears to be not as settled in functional logic languages as it is in purely functional ones. To see this, let us now consider another definition which employs a recursive let-expression:

```
let len = 0 ? (1+len) in len
```

One could reasonably expect that the semantics of this expression is equal to the semantics of the following expression:

```
let ones = [] ? (1:ones) in length ones
```

$$[\![\mathbf{let}\ \overline{x_n :: \tau_n = e_n}\ \mathbf{in}\ e]\!]_{\theta,\sigma} = \bigsqcup_{(\overline{\mathbf{t}_n}) \in \mathbf{T}_{\overline{x_n = e_n}}} [\![e]\!]_{\theta,\sigma[\overline{x_n \mapsto \mathbf{t}_n}]}$$

with $\mathbf{T}_{\overline{x_n=e_n}}$ the set of the minimal tuples $(\overline{\mathbf{t}_n})$ that fulfill the following requirements:

$$\mathbf{t}_1 \in \max(([\![e_1]\!]_{\theta,\sigma[\overline{x_n \mapsto \mathbf{t}_n}]})_\perp), \ldots, \mathbf{t}_n \in \max(([\![e_n]\!]_{\theta,\sigma[\overline{x_n \mapsto \mathbf{t}_n}]})_\perp)$$

Figure 4: Attempt at dealing with recursive let-expressions denotationally

But for `len`, different Curry implementations behave differently. While KiCS indeed yields the result 0 and then runs out of memory (as would be expected by analogy with the `ones` example), PAKCS starts enumerating all natural numbers.

We see that setting up and implementing a formal semantics for a functional logic language in the presence of recursive let-expressions is not likely to be straightforward. All the more, creating a formal semantics in a denotational, functional style promises to help uncovering and handling complex effects such as those described above. We had indeed originally (Christiansen et al. 2011) allowed recursive let-expressions in our denotational semantics, and replaced the last rule in Figure 3 by the one given in Figure 4. That allowed us to explain the difference between `ones` defined via a recursive let-expression and via a top-level definition. In particular, we had (for comparison with the calculation at the end of Section 4):

$$\begin{aligned}
&[\![\mathbf{let}\ \mathit{ones} = \mathsf{Nil}_{\mathsf{Nat}}\ ?\ \mathsf{Cons}(1, \mathit{ones})\ \mathbf{in}\ \mathit{ones}]\!]_{\emptyset,\emptyset}\\
&= \bigsqcup_{\mathbf{t} \in \mathbf{T}_{\mathit{ones} = \mathsf{Nil}_{\mathsf{Nat}} ? \mathsf{Cons}(1,\mathit{ones})}} [\![\mathit{ones}]\!]_{\emptyset,\{\mathit{ones} \mapsto \mathbf{t}\}}\\
&\quad \text{with } \mathbf{T}_{\mathit{ones} = \mathsf{Nil}_{\mathsf{Nat}} ? \mathsf{Cons}(1,\mathit{ones})}\\
&\qquad = \min\{\mathbf{t} \mid \mathbf{t} \in \max(([\![\mathsf{Nil}_{\mathsf{Nat}}\ ?\ \mathsf{Cons}(1, \mathit{ones})]\!]_{\emptyset,\{\mathit{ones} \mapsto \mathbf{t}\}})_\perp)\}\\
&\qquad = \min\{\mathbf{t} \mid \mathbf{t} \in \max((\{[\,]\} \cup \begin{cases} \{\perp : \perp, 1 : \perp\} & \text{if } \mathbf{t} = \perp \\ \bigsqcup_{\mathbf{t}' \in (\{\mathbf{t}\}\downarrow)_\perp} \{\perp : \mathbf{t}', 1 : \mathbf{t}'\} & \text{otherwise} \end{cases})_\perp)\}\\
&\qquad = \min\{\mathbf{t} \mid \mathbf{t} \in \max((\{[\,]\} \cup \{1 : \mathbf{t}\}\downarrow)_\perp)\}\\
&\qquad = \min\{[\,], 1 : 1 : \cdots\}\\
&= \{[\,], 1 : 1 : \cdots\}\downarrow,
\end{aligned}$$

where the crucial step was to realize that

$$\{\mathbf{t} \mid \mathbf{t} \in \max((\{[\,]\} \cup \{1 : \mathbf{t}\}\downarrow)_\perp)\} = \{[\,], 1 : 1 : \ldots\}.^{5}$$

We even conjectured that Theorem 5.4 and Conjecture 5.5, and thus Conjecture 5.3, hold in the presence of recursive let-expressions with that semantics (while allowing recursive heaps in Definition 5.1), but inspired by an example of Schmidt-Schauß et al. (2009) we have since found that the proposed denotational treatment of recursive let-expressions is not consistent with the operational behavior. More precisely, the denotation of expressions that contain recursive let-expressions may consist of more results than it is supposed

[5] No other choice for $\mathbf{t}$ satisfies the requirement $\mathbf{t} \in \max((\{[\,]\} \cup \{1 : \mathbf{t}\}\downarrow)_\perp)$. For example, the singleton list $1 : [\,]$ is not in $(\{[\,]\} \cup \{1 : 1 : [\,]\}\downarrow)_\perp$, and the partial list $1 : \perp$ is in $(\{[\,]\} \cup \{1 : 1 : \perp\}\downarrow)_\perp$, but not maximal in it.

to. Let us demonstrate this by considering the following expression, which is actually very similar to an example of Braßel and Huch (2007b, to show that rule (VarExp) of Albert et al. (2005) is inappropriate):

$$\mathbf{let}\ b = \mathsf{True}\ ?\ \mathbf{case}\ b\ \mathbf{of}\ \{\mathsf{True} \to \mathsf{False}; \mathsf{False} \to \mathsf{False}\}\ \mathbf{in}\ b$$

Evaluating this expression in KiCSi yields True as first result. Asking for more results leads to nontermination. This is the intended behavior in the presence of call-time choice: since b is a variable it can only be bound to one deterministic choice. Therefore, the evaluation of the term above should yield the union of the results of the evaluation of $\mathbf{let}\ b = \mathsf{True}\ \mathbf{in}\ b$ and $\mathbf{let}\ b = \mathbf{case}\ b\ \mathbf{of}\ \{\mathsf{True} \to \mathsf{False}; \mathsf{False} \to \mathsf{False}\}\ \mathbf{in}\ b$, i.e., denotationally the union of $\{\mathit{True}\}$ and $\emptyset$. But the denotational semantics we proposed additionally yields the result *False*. Let us examine the corresponding calculation in a bit more detail:

$$\begin{aligned}
&[\![\mathbf{let}\ b = \mathsf{True}\ ?\ \mathbf{case}\ b\ \mathbf{of}\ \{\mathsf{True} \to \mathsf{False}; \mathsf{False} \to \mathsf{False}\}\ \mathbf{in}\ b]\!]_{\emptyset,\emptyset} \\
= &\bigsqcup\nolimits_{\mathbf{t} \in \mathbf{T}_{b=\mathsf{True}?\mathbf{case}\,b\,\mathbf{of}\,\{\mathsf{True}\to\mathsf{False};\mathsf{False}\to\mathsf{False}\}}} [\![b]\!]_{\emptyset,[b\mapsto\mathbf{t}]} \\
&\text{with } \mathbf{T}_{b=\mathsf{True}?\mathbf{case}\,b\,\mathbf{of}\,\{\mathsf{True}\to\mathsf{False};\mathsf{False}\to\mathsf{False}\}} \\
&\qquad = \min\{\mathbf{t} \mid \mathbf{t} \in \\
&\qquad\qquad \max(([\![\mathsf{True}\ ?\ \mathbf{case}\ b\ \mathbf{of}\ \{\mathsf{True} \to \mathsf{False}; \mathsf{False} \to \mathsf{False}\}]\!]_{\emptyset,[b\mapsto\mathbf{t}]})_\perp)\} \\
&\qquad = \min\{\mathbf{t} \mid \mathbf{t} \in \max((\{\mathit{True}\} \cup \begin{cases}\emptyset & \text{if } \mathbf{t} = \perp \\ \{\mathit{False}\} & \text{otherwise}\end{cases})_\perp)\} \\
&\qquad = \{\mathit{True}, \mathit{False}\} \\
= &\{\mathit{True}, \mathit{False}\}
\end{aligned}$$

The problem becomes visible best in the third-last line of the calculation. Let us assume that the result that originates from the non-recursive part of the right-hand side of the variable binding, namely $\{\mathit{True}\}$, is not present. In this case possible values for $\mathbf{t}$, over which to minimize, are exactly $\perp$ and *False*, because $\perp \in \max(\emptyset_\perp)$ and $\mathit{False} \in \max(\{\mathit{False}\}_\perp)$, but $\mathit{True} \notin \max(\{\mathit{False}\}_\perp)$. After minimization only $\perp$ remains. If we, however, reconsider the original situation where $\{\mathit{True}\}$ *is* present, $\perp$ does not even take part in the minimization, because $\perp \notin \max((\{\mathit{True}\} \cup \emptyset)_\perp)$. Due to $\mathit{True}, \mathit{False} \in \max((\{\mathit{True}\} \cup \{\mathit{False}\})_\perp)$ we now have to minimize over the set $\{\mathit{True}, \mathit{False}\}$ rather than over the set $\{\perp, \mathit{False}\}$, and thus *False* "survives".

Contrary to that denotational semantics, the natural semantics does yield the same results as KiCSi for the above expression, as we will show now. To save space we abbreviate the term $\mathbf{case}\ b\ \mathbf{of}\ \{\mathsf{True} \to \mathsf{False}; \mathsf{False} \to \mathsf{False}\}$ by $\mathbf{seq}^b_{\mathsf{False}}$. The following derivation is the only successful derivation for the expression in question:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\emptyset : \mathsf{True} \Downarrow \emptyset : \mathsf{True}}\ (\text{Val})}{\emptyset : \mathsf{True}\ ?\ \mathbf{seq}^b_{\mathsf{False}} \Downarrow \emptyset : \mathsf{True}}\ (\text{Or}_1)}{\{b \mapsto \mathsf{True}\ ?\ \mathbf{seq}^b_{\mathsf{False}}\} : b \Downarrow \{b \mapsto \mathsf{True}\} : \mathsf{True}}\ (\text{Lookup})}{\emptyset : \mathbf{let}\ b = \mathsf{True}\ ?\ \mathbf{seq}^b_{\mathsf{False}}\ \mathbf{in}\ b \Downarrow \{b \mapsto \mathsf{True}\} : \mathsf{True}}\ (\text{Let})$$

Crucially, choosing $(\textsc{Or}_2)$ instead of $(\textsc{Or}_1)$ leads to a partial derivation that cannot be completed:

$$\dfrac{\dfrac{\emptyset : b \Downarrow ??? \qquad ??? \Downarrow ???}{\emptyset : \textbf{case}\ b\ \textbf{of}\ \{\mathsf{True} \to \mathsf{False}; \mathsf{False} \to \mathsf{False}\} \Downarrow}\,(???)}{\emptyset : \mathsf{True}\ ?\ \mathbf{seq}^{b}_{\mathsf{False}} \Downarrow}\,(\textsc{Or}_2)$$

For the rule (???) we could try to choose $(\textsc{LSelect}_1)$, $(\textsc{LSelect}_2)$, $(\textsc{LGuess}_1)$ or $(\textsc{LGuess}_2)$, but in the left branch we would always end up asking the empty heap for the value of b, thus getting stuck.

The example presented above proves that Conjecture 5.3 does not hold in the presence of recursive let-expressions with their proposed denotational treatment (and, more specifically, neither does Conjecture 5.5). From our current perspective that flaw is unfixable in any approach to a set-valued denotational semantics. To define such a semantics for a recursive let-expression it is simply not sufficient to know the sets which would be assigned to the right-hand sides of variable bindings. Instead, it needs to be known wherefrom the elements in such a set arise. And that information is not accessible in general.

We still think that Theorem 5.4 holds even in the presence of recursive let-expressions (as does Lemma 5.9), with our proposed denotational treatment of them, though it is doubtful how useful that is in practice, given that that part of our denotational semantics is not really adequate for full Curry. The fragment that contains only non-recursive let-expressions is still powerful enough to model an interesting part of the language. Hence, our semantics remains a suitable choice for equational reasoning and as a foundation for formally carrying over relational parametricity arguments to functional logic languages.

References

S. Abramsky and A. Jung. Domain theory. In *Handbook of Logic in Computer Science*, pages 1–168. Oxford University Press, 1994.

E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. Operational semantics for declarative multi-paradigm languages. *Journal of Symbolic Computation*, 40(1):795–829, 2005.

B. Braßel and F. Huch. On a tighter integration of functional and logic programming. In *Asian Symposium on Programming Languages and Systems, Proceedings*, LNCS 4807:122–138. Springer-Verlag, 2007a.

B. Braßel and F. Huch. On the tighter integration of functional and logic programming. Technical Report 0710, Department of Computer Science, University Kiel, 2007b.

B. Braßel, S. Fischer, M. Hanus, and F. Reck. Transforming functional logic programs into monadic functional programs. In *Workshop on Functional and (Constraint) Logic Programming 2010, Post-Proceedings*, LNCS 6559:31–48. Springer-Verlag, 2011.

J. Christiansen, D. Seidel, and J. Voigtländer. Free theorems for functional logic programs. In *Programming Languages meets Program Verification, Proceedings*, pages 39–48. ACM Press, 2010.

J. Christiansen, D. Seidel, and J. Voigtländer. An adequate, denotational, functional-style semantics for typed FlatCurry. In *Workshop on Functional and (Constraint) Logic Programming 2010, Post-Proceedings*, LNCS 6559:119–136. Springer-Verlag, 2011.

L. Erkök. *Value Recursion in Monadic Computations.* PhD thesis, OGI School of Science and Engineering, OHSU, 2002.

J.C. González-Moreno, M.T. Hortalá-González, F.J. López-Fraguas, and M. Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming*, 40(1):47–87, 1999.

F.J. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. Equivalence of two formal semantics for functional logic programs. In *Programación y Lenguajes 2006, Proceedings*, ENTCS 188:117–142. Elsevier, 2007.

F.J. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. A fully abstract semantics for constructor systems. In *Rewriting Techniques and Applications, Proceedings*, LNCS 5595:320–334. Springer-Verlag, 2009.

J.M. Molina-Bravo and E. Pimentel. Composing programs in a rewriting logic for declarative programming. *Journal of Theory and Practice of Logic Programming*, 3(2):189–221, 2003.

M. Schmidt-Schauß, E. Machkasova, and D. Sabel. Counterexamples to simulation in non-deterministic call-by-need lambda-calculi with letrec. Technical Report Frank-38, Institute of Computer Science, University Frankfurt, 2009.

H. Søndergaard and P. Sestoft. Non-determinism in functional languages. *The Computer Journal*, 35(5):514–523, 1992.

P. Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture, Proceedings*, pages 347–359. ACM Press, 1989.

M. Walicki and S. Meldal. Algebraic approaches to nondeterminism: An overview. *ACM Computing Surveys*, 29(1):30–81, 1997.

IAI-TR-95-1 Modelle für die Photogrammetrische Gebäudekonstruktion
C. Braun, T.H. Kolbe, F. Lang, W. Schickler, V. Steinhage, A.B. Cremers, W. Förstner and L. Plümer
Januar 1995.

IAI-TR-95-2 The Mobile Robot Rhino
J. Buhmann, W. Burgard, A.B. Cremers, D. Fox, T. Hofmann, F. Schneider, J. Strikos and S. Thrun
Januar 1995.

IAI-TR-95-3 Levesque's Axiomatization of Only Knowing is Incomplete
Joseph Y. Halpern and Gerhard Lakemeyer
Januar 1995.

IAI-TR-95-4 First Specification of Phoenix, an Active Database Programming Language
Ulrike Griefahn
Januar 1995.

IAI-TR-95-5 Extended circle graphs I
Christoph Hundack and Hermann Stamm-Wilbrandt
März 1995.

IAI-TR-95-6 Knowledge-enhanced CO-monitoring in Coal Mines
Wolfram Burgard and Armin B. Cremers and Dieter Fox and Martin Heidelbach and Angelica M. Kappel and Stefan Lüttringhaus-Kappel
März 1995.

IAI-TR-95-7 Pairwise Data Clustering by Deterministic Annealing
Joachim M. Buhmann and Thomas Hofmann
Juni 1995.

IAI-TR-95-8 Penguin - A Portable Environment for a Graphical User Interface
Dieter W. Fellner and Martin Fischer
Juni 1995.

IAI-TR-95-9 A Portable Algorithm for Dynamic Color Management
Martin Fischer
Juni 1995.

IAI-TR-95-10 Distributed CGI^{++} - eine Client/Server-Lösung zur Entwicklung leistungsfähiger Computergraphik-Anwendungen
Dieter W. Fellner and Martin Fischer
Juli 1995.

IAI-TR-95-11 Runtime Type for Class Hierarchies
Martin Fischer
Juli 1995.

IAI-TR-95-12 Reference Pointer for Class Hierarchies
Martin Fischer
August 1995.

IAI-TR-95-13 The Dynamic Window Approach to Collision Avoidance
Dieter Fox, Wolfram Burgard and Sebastian Thrun
August 1995.

IAI-TR-96-1 Estimating the Absolute Position of a Mobil Robot Using Position Probability Grids
Wolfram Burgard, Dieter Fox
Daniel Hennig and Timo Schmidt
Februar 1996.

IAI-TR-96-2 A Deterministic Annealing Framework for Unsupervised Texture Segmentation
Thomas Hofmann, Jan Puzicha and Joachim M. Buhmann
Februar 1996.

IAI-TR-96-3 Map Learning and High-Speed Navigation in RHINO
S. Thrun, A. Bücken, W. Burgard, D. Fox, T. Fröhlinghaus, D. Hennig, T. Hofmann, M. Krell and T. Schmidt
Juli 1996.

IAI-TR-96-4 Belief Revision in a Nonclassical Logic
Gerhard Lakemeyer and Wilfried Lang
Juli 1996.

IAI-TR-96-5 Bildverarbeitung und Mustererkennung
W. Förstner und A.B. Cremers
Juli 1996.

IAI-TR-96-6 Implementing Chimera on Top of an Active Relational Database System
Ulrike Griefahn and Thomas Lemke
Juli 1996.

IAI-TR-2002-9 Planning to Optimize the UMTS Call Set-up for the Execution of Mobile Applications
Roman Englert
Oktober 2002.

IAI-TR-2003-1 Program-independent Composition of Conditional Transformations
Günter Kniesel and Helge Koch
Juni 2003.

IAI-TR-2003-2 The JMangler Tutorial
Günter Kniesel
Juli 2003.

IAI-TR-2003-3 Static Dependency Analysis for Conditional Program Transformations
Günter Kniesel
Juli 2003.

IAI-TR-2003-4 Objektorientierte Datenmodell-Integration für ein offenes, Paläoökologisches Informationssystem
A.B. Cremers, R. Dikau (Hrsg.)
Dezember 2003.

IAI-TR-2003-5 Concepts for the Representation, Storage, and Retrieval of Spatio-Temporal Objects in 3D/4D Geo-Information-Systems
Jörg Siebeck
Dezember 2003.

IAI-TR-2004-1 Modell- und diagnosebasiertes Transformationslernen von symbolischen Plänen für mobile Roboter
Michael Beetz, Thorsten Belker, Armin B. Cremers
Februar 2004.

IAI-TR-2004-2 Unantizipierte Software-Evolution - Abschlussbericht des TAILOR Projekts (DFG-Projekt CR 65/13)
Pascal Costanza, Günter Kniesel, Armin B. Cremers
Oktober 2004.

IAI-TR-2004-3 A Meta Model for AspectJ
YAN Han, Günter Kniesel, Armin B. Cremers
Oktober 2004.

IAI-TR-2004-4 Uniform Genericity for Aspect Languages
Tobias Rho, Günter Kniesel
Dezember 2004.

IAI-TR-2006-1 A Logic Foundation for Program Transformations
Günter Kniesel
Januar 2006.

IAI-TR-2007-2 Globally Optimal Image Segmentation with an Elastic Shape Prior
Thomas Schoenemann, Daniel Cremers
April 2007.

IAI-TR-2007-3 Dual-space Graph Cuts for Motion Layer Decomposition
Thomas Schoenemann and Daniel Cremers
April 2007.

IAI-TR-2007-4 Iterated and Efficient Nonlocal Means for Denoising of Textural Patterns
Thomas Brox, Oliver Kleinschmidt, and Daniel Cremers
August 2007.

IAI-TR-2007-5 Design and Implementation of a Workflow Engine
Sebastian Bergmann
September 2007.

IAI-TR-2007-6 On Local Region Models and the Statistical Interpretation of the Piecewise Smooth Mumford-Shah Functional
Thomas Brox and Daniel Cremers
September 2007.

IAI-TR-2007-7 Time-based Triggers for SQL: A Lingering Issue Revisited
Andreas Behrend, Christian Dorau, and Rainer Manthey
November 2007.

IAI-TR-2007-8 High Resolution Motion Layer Decomposition using Dual-space Graph Cuts
Thomas Schoenemann and Daniel Cremers
Dezember 2007.

IAI-TR-2007-9 Matching Non-rigidly Deformable Shapes Across Images: A Globally Optimal Solution
(DFG-ProjekT CR 250/1-1)
Thomas Schoenemann and Daniel Cremers
Dezember 2007.

IAI-TR-2007-10 Globally Optimal Shape-based Tracking in Real-time
(DFG-ProjekT CR 250/1-1)
Thomas Schoenemann and Daniel Cremers
Dezember 2007.

IAI-TR-2008-1 Software Security Metrics for Malware Resilience
Hanno Langweg
Februar 2008.

IAI-TR-2008-2 Rule-Based Programming
Günter Kniesel, Jorge Sousa Pinto
Juni 2008.

IAI-TR-2008-3 Efficient Planar Graph Cuts with Applications in Computer Vision
Frank R. Schmidt, Eno Töppe and Daniel Cremers
Juni 2008.

IAI-TR-2008-4 The Elastic Ratio: Introducing Curvature into Ratio-based Globally Optimal Image Segmentation
Thomas Schoenemann, Simon Masnou and Daniel Cremers
Dezember 2008.

IAI-TR-2008-5 A convex approach for computing minimal partitions
Antonin Chambolle, Daniel Cremers, Thomas Pock
November 2008.

IAI-TR-2009-1 Anisotropic Minimal Surfaces for Integration of Multiview Stereo and Normal Information
Kalin Kolev, Thomas Pock, Daniel Cremers
Juni 2009.

IAI-TR-2009-2 Witnessing Patterns: A Data Fusion Approach to Design Pattern Detection
Alexander Binun, Günter Kniesel
Januar 2009.

IAI-TR-2009-3 DPDX - A Common Exchange Format for Design Pattern Detection Tools
Günter Kniesel, Alexander Binun, Péter Hegedűs, Lajos Jenő Fülöp, Alexander Chatzigeorgiou, Yann-Gaël Guéhéneuc and Nikolaos Tsantalis
Oktober 2009.

IAI-TR-2009-4 Moment Constraints in Convex Relaxation Methods for Shape Optimization
Daniel Cremers
September 2009.

IAI-TR-2010-1 3D-Objekterkennung in dynamischen Szenen
J. Behley, V. Steinhage, A.B. Cremers
November 2010.

IAI-TR-2011-1 An Adequate, Denotational, Functional-Style Semantics for Typed FlatCurry without Letrec
Jan Christiansen, Daniel Seidel, and Janis Voigtländer
März 2011.