A Semantics for Weakly Encapsulated Search in Functional Logic Programs

Jan Christiansen
Institut für Informatik
University of Kiel, Germany
jac@informatik.uni-kiel.de

Michael Hanus
Institut für Informatik
University of Kiel, Germany
mh@informatik.uni-kiel.de

Fabian Reck
Institut für Informatik
University of Kiel, Germany
fre@informatik.uni-kiel.de

Daniel Seidel
Institut für Informatik
University of Bonn, Germany
ds@iai.uni-bonn.de

ABSTRACT

Encapsulated search is a key feature of (functional) logic languages. It allows the programmer to access and process different results of a non-deterministic computation within a program. Unfortunately, due to advanced operational features (lazy evaluation, partial values, infinite structures), there is no straightforward definition of the semantics of encapsulated search in functional logic languages. As a consequence, various proposals and implementations are available but a rigorous definition covering all semantical aspects does not exist. In this paper, we analyze the requirements of encapsulated search in a functional logic language like Curry and provide a comprehensive definition that covers weak encapsulation, a modular form of encapsulation, as well as nested applications of search operators. We set up a denotational semantics that distinguishes non-termination and different levels of failures in a computation. The semantics is also the basis of a practical implementation of search operators in the functional logic language Curry.

Categories and Subject Descriptors

D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*Semantics*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Control structures*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Denotational semantics*

General Terms

Algorithms, Design, Languages, Theory

Keywords

Functional logic programming languages, rewrite systems, non-determinism, encapsulation

PPDP '13, September 16 - 18 2013, Madrid, Spain. Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2154-9/13/09 ...$15.00.
http://dx.doi.org/10.1145/2505879.2505896.

1. INTRODUCTION

Functional logic languages combine the most appealing features of the functional world, like algebraic data types, higher-order and lazy evaluation, with features from the logical world, like non-determinism and free variables (see [4, 14] for recent surveys). The functional logic language Curry [18], which is considered in this paper, is based on a demand-driven (lazy) evaluation strategy that is optimal for a wide class of programs [3]. Due to its logic programming features, an expression in Curry might have multiple results that are evaluated non-deterministically. In typical Curry systems these results are shown in a read-eval-print-loop (REPL): the user enters an expression and the system displays the computed values in some order determined by the search strategy. For instance, the Curry system KiCS2 [6] supports depth-first, breadth-first, iterative deepening, or parallel search strategies.

In actual applications, however, the non-deterministic computations should be *encapsulated*: instead of processing (and eventually printing) all results of a subcomputation, one wants to select a single result, e.g., the shortest itinerary among all possible itineraries between two cities [5]. Therefore, many non-deterministic languages provide primitives to compute the set of all results of a computation. In this way results can be related, e.g., by computing the smallest of the results or by accumulating them. For instance, Prolog has a primitive `findall` [23] to compute the list of all answers to a goal.

In this work we specify the semantics of an encapsulation primitive **allValues** so that $\mathbf{allValues}(e)$ denotes the set of all values the expression e can evaluate to. However, the precise meaning of this primitive is not obvious due to the combination of laziness and non-determinism:

1. Does the result set contain fully evaluated expressions or only head normal forms?
2. Which non-deterministic computations are encapsulated? For instance, consider the expression

$$\mathbf{let}\ x = e_1\ \mathbf{in}\ \mathbf{allValues}(e_2)$$

where x occurs in e_2 and both, e_1 and e_2, evaluate non-deterministically to multiple results. Is the non-determinism of e_1 encapsulated by $\mathbf{allValues}(e_2)$?

3. If there are nested applications, as in

$$\mathbf{allValues}(e_1\ \mathbf{allValues}(e_2)\ e_3)$$

which non-deterministic computations are encapsulated by which search operator? In particular, if some subcomputation fails, how does it affect the encapsulation?

Lazy evaluation complicates the answers to these questions. For instance, the evaluation of e_2 in the expressions above might trigger the evaluation of some subexpression of e_1 caused by shared variables. The various proposals for encapsulating search in functional logic languages (e.g., [5, 8, 9, 17, 20, 21]) together with semantically different implementations in various Curry systems (as discussed later) demonstrate that there are no clear answers to these questions.

In order to specify the intended results of expressions and to be independent of implementation details, we propose a denotational semantics for the encapsulation primitive **allValues**, inspired by a denotational semantics without capsules [11]. As we will see, a precise description of nested capsules (item 3 above) requires the introduction of different failure values in the semantical domain so that one can distinguish between non-termination and finite failures occuring in different nesting levels. This distinction is not made by semantical descriptions that do not regard capsules (e.g., [12]).

Our semantics helps users of Curry to understand the detailed meaning and semantical consequences of such a primitive, e.g., we can show that particular program transformations used in the implementation of purely functional languages are no longer valid in the presence of such an encapsulation primitive. Furthermore, our semantics is the basis for implementing advanced search operators in the recent Curry implementation KiCS2 [6, 16].

The main focus of this paper is to provide a formal foundation for encapsulated search in a functional logic language. We introduce a kernel language for Curry programs that features primitives to calculate and test the set of non-deterministic results of a computation (Section 2). After discussing the intended behavior of encapsulated search (Section 3), we define its denotational semantics (Section 4). Section 5 shows our semantics at work. In Section 6 we sketch an implementation of this semantics before we conclude with a discussion on related work in Section 7.

2. (FLAT)CURRY

The syntax of Curry is very similar to the syntax of the functional language Haskell [25]. In contrast to Haskell, Curry additionally provides free variables and the means to introduce non-determinism. However, many tools that process Curry programs, such as compilers and analyzers, actually work on a kernel language, called *FlatCurry*, into which all Curry programs can be translated by eliminating syntactic sugar from source programs. Christiansen et al. [11] present a denotational semantics for a variant of this intermediate language, called TFLC.

On the one hand, we extend TFLC with a construct for encapsulation, a primitive data type for sets and operations on sets. On the other hand, we exclude polymorphic types and restrict let-expressions to bind only a single variable. These restrictions are reasonable since polymorphism does not add extra insights and non-recursive let-expressions with more than one binding can easily be split up into a series of bindings. Furthermore, Christiansen et al. [11] point out that recursive let-expressions cannot be handled by a compositional semantics like the one we head for. We could also omit let-expressions completely because they can be simulated by function calls, but they are useful for example calculations.

The syntax of our extended kernel language TFLC^e is defined in Figure 1. Similarly to TFLC, we consider a language with just a few, concrete data types that can be easily extended, e.g., with other types and primitive operations. In Figure 1, τ denotes a type,

τ	$::=$	$\mathsf{Nat} \mid \mathsf{Bool}$	(base types)
	$\mid$	$[\tau]$	(list type)
	$\mid$	$\tau_1 \to \tau_2$	(function type)
	$\mid$	$\{\tau\}$	(set type)
P	$::=$	$D\ P \mid \epsilon$	
D	$::=$	$f :: \tau;\ f(\overline{x_n}) = e$	
e	$::=$	x	(variable)
	$\mid$	n	(number)
	$\mid$	$\mathsf{True} \mid \mathsf{False}$	(data constructors)
	$\mid$	$\mathsf{Nil}_\tau \mid \mathsf{Cons}(e_1, e_2)$	
	$\mid$	$e_1 + e_2$	(primitive operation)
	$\mid$	f	(defined function)
	$\mid$	$\mathbf{apply}(e_1, e_2)$	(application)
	$\mid$	$\mathbf{unknown}_\tau$	(unknown value)
	$\mid$	$e_1\ ?\ e_2$	(non-deterministic choice)
	$\mid$	$\mathbf{failed}_\tau$	(failure)
	$\mid$	$\mathbf{case}\ e\ \mathbf{of}$	
		$\{\mathsf{True} \to e_1$	(case on Booleans)
		$;\mathsf{False} \to e_2\}$	
	$\mid$	$\mathbf{case}\ e\ \mathbf{of}$	
		$\{\mathsf{Nil} \to e_1$	(case on lists)
		$;\mathsf{Cons}(x, xs) \to e_2\}$	
	$\mid$	$\mathbf{let}\ x :: \tau = e_1\ \mathbf{in}\ e_2$	(local definition)
	$\mid$	$\mathbf{allValues}_\tau(e)$	(encapsulated search)
	$\mid$	$\mathbf{size}_\tau(e)$	(size of a set)
	$\mid$	$\mathbf{isEmpty}_\tau(e)$	(emptiness test on a set)
	$\mid$	$\mathbf{minimum}_\tau(e)$	(minimum of a set)

Figure 1: Syntax of the kernel language TFLC^e

P a program, D a function definition, and e an expression. Furthermore, x stands for an expression variable and n for a natural number. We call a type τ *simple* iff it is constructed without the use of $\to$. A bar, as in $\overline{x_n}$, denotes the sequence $x_1, \ldots, x_n$. An expression of the form $(e_1\ ?\ e_2)$ denotes a non-deterministic choice between e_1 and e_2, $\mathbf{apply}(e_1, e_2)$ expresses the application of e_1 to e_2 (and covers higher-order features), $\mathbf{unknown}_\tau$ the non-deterministic choice between all values of type τ (i.e., a free variable in the sense of logic programming) and $\mathbf{failed}_\tau$ a finite failure. However, some restrictions apply: the types of free variables have to be simple and must not contain any set types ($\{\tau\}$) and, as already told, let-expressions have to be non-recursive.

In order to deal with encapsulated search, we extend TFLC with the primitive **allValues** and the set operations **size**, **isEmpty** and **minimum**. Note that **allValues** yields a set rather than any ordered structure to hide the order imposed by the search strategy [8]. We exclusively allow sets of simple types, since implementations are not able to test functions for (extensional) equality. Therefore, **allValues** may only be applied to expressions of simple type. Further operations on sets can be added but are omitted here for simplicity. Such operations have to respect the set properties, especially the internal ordering must not leak through the abstraction. E.g., it is not possible to obtain a “first” object from the set without sorting the set or escape to an IO context where we can ask an “oracle” for a virtually arbitrary ordering that coincidentally matches the internal one.

The typing rules for the additional primitives of TFLC^e are given by Figure 2. The first rule states that the encapsulation primitive that is applied to an expression of type τ yields a result of type $\{\tau\}$. The other rules state that the set operations can only be ap-

$$\frac{\Gamma \vdash e :: \tau \quad \tau \text{ is simple}}{\Gamma \vdash \textbf{allValues}_\tau(e) :: \{\tau\}} \qquad \frac{\Gamma \vdash e :: \{\tau\}}{\Gamma \vdash \textbf{size}_\tau(e) :: \mathsf{Nat}}$$

$$\frac{\Gamma \vdash e :: \{\tau\}}{\Gamma \vdash \textbf{isEmpty}_\tau(e) :: \mathsf{Bool}} \qquad \frac{\Gamma \vdash e :: \{\tau\}}{\Gamma \vdash \textbf{minimum}_\tau(e) :: \tau}$$

Figure 2: Typing rules for encapsulation and set operations

plied to sets and yield a natural number, a Boolean, or a value with a type that corresponds to the elements in the set, respectively. The typing context Γ provides type information for unbound variables. Typing rules for the remaining language constructs can be found in [11]. In example programs, we often omit type subscripts if they can be easily determined by the context. As a further notational simplification, we write applications of an expression f to an expression e as $f(e)$ instead of $\textbf{apply}(f, e)$ (and similarly for more than one argument).

3. REQUIREMENT ANALYSIS

As mentioned above, there are various proposals to encapsulate non-deterministic computations in functional logic programs, e.g., [5, 8, 9, 17, 20, 21]. The most recent approach [5] proposes *set functions*. With set functions, for every function f in a Curry program, there is a function f_S that yields the set of all results of f applied to some deterministic input. Set functions encapsulate only the non-determinism that is introduced by the function's definition but not the non-determinism brought in via arguments. For instance, consider the operation *inc12* defined as

$$\begin{aligned} \mathit{inc12} &:: \mathsf{Nat} \rightarrow \mathsf{Nat} \\ \mathit{inc12}(x) &= \ (x+1) \ \ ? \ \ (x+2) \end{aligned}$$

Then $\mathit{inc12}_S(1)$ evaluates to some internal representation of the set $\{2, 3\}$, i.e., the non-determinism caused by *inc12* is encapsulated into a set. However, the expression $\mathit{inc12}_S(1 \ ? \ 5)$ evaluates to two different sets $\{2, 3\}$ and $\{6, 7\}$ due to its non-deterministic argument, i.e., the non-determinism caused by the argument is not encapsulated.

In contrast to many other approaches to encapsulated search (see [8] for a detailed discussion), the result of a set function does not depend on the evaluation steps that might be performed by the context of the encapsulated expression—a modular and, thus, desirable property for declarative programming. Therefore, our approach should conform with the basic ideas of set functions. Using our primitive $\textbf{allValues}$, for every Curry function $f(x_1, \ldots, x_n) = e$ we can define the corresponding set function as

$$f_S(x_1, \ldots, x_n) = \textbf{allValues}(e)$$

However, as discussed below, the semantics of set functions that is presented in [5] is underspecified in the presence of finite failures and nested applications of set functions. Thus, a proposed implementation [7] of set functions does not yield the intended results for the motivating example of [5].

In order to obtain a comprehensive and reasonable definition of the meaning of $\textbf{allValues}$, in this section we consider the requirements for encapsulating non-deterministic computations.

3.1 Normal Form Encapsulation

Consider the following TFLC^e program where *coin* denotes a non-deterministic expression:

$$\begin{aligned} \mathit{coin} &:: \mathsf{Nat} \\ \mathit{coin}() &= 0 \ ? \ 1 \\ \mathit{numCoinLists} &:: \mathsf{Nat} \\ \mathit{numCoinLists}() &= \textbf{size}(\textbf{allValues}(\mathsf{Cons}(\mathit{coin}, \mathsf{Nil}))) \end{aligned}$$

What is the intended result of a call to *numCoinLists*? There are two possible choices. If $\textbf{allValues}$ does not evaluate the elements of the list $\mathsf{Cons}(\mathit{coin}, \mathsf{Nil})$ (since the values of the individual elements are not demanded), the non-determinism in *coin* is not uncovered, $\textbf{allValues}$ returns a set with only one list and $\textbf{size}$ returns 1. If $\textbf{allValues}$ evaluates its argument completely (similarly to the top-level REPL of a Curry system), we get a set with the elements $\mathsf{Cons}(0, \mathsf{Nil})$ and $\mathsf{Cons}(1, \mathsf{Nil})$, and, thus, $\textbf{size}$ returns 2.

The first option is called *head normal form encapsulation* and proposed in [9]. All elements of the set that $\textbf{allValues}$ returns are in head normal form (i.e., without a defined function as the outermost symbol) but not necessarily in normal form, i.e., completely evaluated. This form of encapsulation implies that the result of an encapsulated expression may still contain nested non-determinism, as seen by the example.

The second option is called *normal form encapsulation* and proposed in [5]. In this case the argument to $\textbf{allValues}$ is fully evaluated, i.e., the set contains only total values. Thus, all non-determinism is capsuled.

For our semantics, we choose the second option since this conforms with the "REPL view of non-determinism": the values in an encapsulation set are also the values shown by the REPL of a Curry system. This option is also used for set functions [5]. Moreover, head normal form encapsulation might cause weird effects in combination with sharing when further evaluation of encapsuled entries is enforced. This could be possible if specific elements from a set are processed.

3.2 Weak Encapsulation

A key feature of declarative languages is that the value of an expression depends solely on the values of its sub-expressions. In particular, the evaluation of a sub-expression does not influence the result of another sub-expression. This property is most important in conjunction with lazy evaluation where the order of evaluation is not easily predictable by the user. Therefore, the demand that $\textbf{allValues}$ should retain this property is consistent.

Braßel et al. [8] investigate various approaches to encapsulated search. They distinguish two concepts, strong encapsulation and weak encapsulation. These concepts differ when non-deterministic expressions are introduced outside an encapsulation primitive but evaluated inside. For example, consider the following TFLC^e program:

$$\begin{aligned} \mathit{coin} &:: \mathsf{Nat} \\ \mathit{coin} &= 0 \ ? \ 1 \\ \mathit{allCoins} &:: \{\mathsf{Nat}\} \\ \mathit{allCoins} &= \textbf{let } x :: \mathsf{Nat} = \mathit{coin} \textbf{ in allValues}(x) \end{aligned}$$

The operation *allCoins* contains a non-deterministic sub-expression *coin* that is bound to x. Since *coin* textually occurs outside of the encapsulation primitive $\textbf{allValues}$, we will consider it as *introduced outside*. With *strong encapsulation*, any non-determinism that is evaluated by an encapsulation primitive is encapsulated, no matter where it is introduced. Thus, *allCoins* would yield the set $\{0, 1\}$ w.r.t. strong encapsulation. The Curry system PAKCS [15] implements this kind of encapsulation.

In contrast, *weak encapsulation* [8] only encapsulates the non-determinism that is introduced inside the encapsulation primitive, e.g., the non-determinism in the expression $\mathbf{allValues}(\mathit{coin})$ is considered to be introduced inside. In the case of weak encapsulation, *allCoins* non-deterministically yields one of the sets $\{0\}$ or $\{1\}$. The Münster Curry Compiler (MCC) [22] provides a primitive `findall` that implements weak encapsulation. However, the function **allValues** cannot be defined by means of this primitive.[1]

Braßel et al. [8] show that with strong encapsulation the evaluation of a shared subexpression in the context can influence the results of an encapsulation primitive. While they restrict the use of encapsulation to prevent this non-declarative behavior, we opt to implement weak encapsulation. Again, our choice is in line with set functions [5].

3.3 Completeness of Results

Encapsulation is meant to provide access to the set of all results of an encapsulated expression. Thus, if v is a value of some expression e, then we have $v \in S$ for some result S of $\mathbf{allValues}(e)$ and vice versa. Furthermore, we want two different results of an expression e to be in the same result S of $\mathbf{allValues}(e)$, if and only if they only differ by different choices made for non-determinism that was introduced inside e. For example, the expression *coin* yields the results 0 and 1. Therefore, there has to be a result of $\mathbf{allValues}(\mathit{coin})$ that contains 0 and a result that contains 1. No other elements are allowed. Since the non-determinism is introduced inside **allValues**, both elements have to be in the same result set. Thus, $\{0, 1\}$ is the only valid result of $\mathbf{allValues}(\mathit{coin})$. However, if we consider the expression

$$\mathbf{let}\ x :: \mathsf{Nat} = \mathit{coin}\ \mathbf{in}\ \mathbf{allValues}(x)$$

there also has to be a result that contains 0 and a result that contains 1. Furthermore, since the non-determinism is introduced outside of **allValues**, 0 and 1 need to be in different result sets. Thus, $\{0\}$ and $\{1\}$ are the only valid results of

$$\mathbf{let}\ x :: \mathsf{Nat} = \mathit{coin}\ \mathbf{in}\ \mathbf{allValues}(x)$$

This requirement is also formally stated for set functions in [5].

3.4 Finite Failures

So far, the intended meaning of our primitive **allValues** is not completely specified. We stated which results have to reside in the same sets, yet sets may also be empty. Empty result sets are of special interest because they allow for programming with *negation as failure*, similarly to logic programming [20]. Consider the following TFLC^e program:

$$\begin{aligned}
\mathit{nilP} &:: [\mathsf{Nat}] \to \mathsf{Bool}\\
\mathit{nilP}(x) &= \mathbf{case}\ x\ \mathbf{of}\\
&\qquad \{\mathsf{Nil} \to \mathsf{True}\\
&\qquad ; \mathsf{Cons}(y, ys) \to \mathbf{failed}_{\mathsf{Bool}}\}\\
\mathit{consP} &:: [\mathsf{Nat}] \to \mathsf{Bool}\\
\mathit{consP}(z) &= \mathbf{isEmpty}(\mathbf{allValues}(\mathit{nilP}(z)))
\end{aligned}$$

The predicate *nilP* yields True if its argument is the empty list and fails otherwise. The function *consP* is intended to yield True exactly if its argument is a non-empty list. It is implemented by means of negation as failure using the predicate *nilP*. If *nilP* is applied to a non-empty list, it fails and, hence, the result of the encapsulated expression is the empty set so that *consP* yields True.

[1]More precisely, the function **allValues** cannot be defined as `allValues e = findall (\x -> x =:= e)` since the non-determinism in the argument would not be encapsulated due to weak encapsulation.

What happens if *consP* is applied to an expression *fl* whose evaluation fails? In a language with a demand-driven evaluation strategy, *fl* is not immediately evaluated but only when pattern matching is performed in the body of *nilP*. This leads to two options:

1. Since the evaluation of $\mathit{nilP}(\mathit{fl})$ fails, the encapsulation primitive returns an empty set and, therefore, the evaluation of $\mathit{consP}(\mathit{fl})$ yields True.
2. Since the failing expression *fl* has been created outside the encapsulated expression, its failure is not covered by the operation **allValues** so that the evaluation of $\mathit{consP}(\mathit{fl})$ fails.

   Thus, failures of expressions created outside of an encapsulation primitive are not covered by **allValues**, similarly to the behavior regarding non-determinism.

The first option leads to the problem that it is hard to understand when a failure introduced outside an encapsulated expression will result in an empty value set for the capsule, since it is very difficult to reason about the control flow in a lazy language. Therefore, we prefer the second option.

Consequently, failures of expressions created outside **allValues** should not influence the result of the encapsulation, i.e., we have to ignore such "outside" failures in a capsule—even when their value is demanded in the encapsulated expression. For instance, consider the following expression:

$$\mathbf{let}\ x :: \mathsf{Bool} = \mathbf{failed}_{\mathsf{Bool}}\ \mathbf{in}\ \mathbf{allValues}(x\ ?\ \mathsf{True})$$

The result of this expression should be the set $\{\mathsf{True}\}$ rather than a failure. Otherwise, the property of the completeness of results would be violated: the value True is a result of $x\ ?\ \mathsf{True}$, independently of the value of x, and, thus, it should appear in a result set of $\mathbf{allValues}(x\ ?\ \mathsf{True})$. Note that this would be different in a (Prolog-like) strict language where the strict evaluation of x would lead to a failure before evaluating the encapsulated expression.

Note that [5] does not specify the handling of failures, although it contains an example for programming with negation as failure that only works with our interpretation of failures in encapsulated computations. A precise definition of this interpretation of failures demands for a sophisticated semantics where different kinds of failures can be distinguished. We introduce this kind of semantics in the following section. As a positive side effect, we also specify the meaning of nested applications of search primitives, which is practically relevant but has not been formally covered in previous approaches for weak encapsulation in non-strict functional logic languages.

4. SEMANTICS

Christiansen et al. [11] present a set-based denotational semantics for FlatCurry. We present a slightly modified version of this semantics and extend it with constructs for encapsulation. The semantics is based on a multialgebraic view on functions. That is, functions map single elements to sets of elements. This view models the call-time choice semantics [19] for non-deterministic operations as used in contemporary functional logic languages. Furthermore, it closely corresponds to the CRWL [12] approach, which is a well established logical foundation for functional logic languages. The denotational semantics is advantageous for our setting in comparison with CRWL because expressions in the denotational semantics are already set-valued whereas CRWL only states how expressions can be rewritten to obtain valid results.

4.1 Semantics of Types

Curry provides angelic ("don't know") non-determinism, which is modeled by the use of the Hoare powerdomain. We restrict ourselves to continuous directed-complete partial orders (dcpos) as domains. The Hoare powerdomain of a dcpo is the complete lattice of all its non-empty Scott-closed subsets. For theoretical background, consult the survey on domain theory from Abramsky and Jung [1], in particular Section 6.2, Theorem 6.2.13. For a continuous dcpo $\mathbf{D} = (D, \sqsubseteq)$, we denote its Hoare powerdomain by $\mathcal{P}_H(\mathbf{D})$. Then infimum and supremum of $M \subseteq \mathcal{P}_H(\mathbf{D})$ are defined by:

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

In our calculus, we interpret types as follows.

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

This semantics of types slightly differs from the one introduced in [11]. First, we need no extra environment, because we abstain from polymorphism as mentioned before. Second, we directly add a $\perp$-element as least element to each dcpo via the lifting operator $(\cdot)_\perp$, which is technically beneficial later on and is more similar to [1]. Regarding the order, the semantics of Bool and Nat are sets with the discrete order and $\perp$ as least element. The semantics of lists is given as least fixpoint (*lfp*). Note that the entries of a list are elements (rather than sets of elements), and, as we are modelling a non-strict language, $\perp$ is a valid entry as well. The order on lists is given by element-wise comparison. The order on the function space is point-wise and the functions need to be Scott-continuous (i.e., monotone and preserving supremas of directed sets) to ensure that the function space itself is a continuous dcpo.

We use $\perp$ to model non-termination but do not use it to model finite failures because the semantics of the encapsulation primitive is supposed to distinguish non-termination from finite failure. Note that we have to distinguish failures that are introduced outside an encapsulated expression from those introduced inside, as required in Section 3.4. In order to model nested encapsulations with different failure levels, we introduce new values for failures from different layers of encapsulation in our semantical domains. These values are represented as $\mathbf{F}_i$ where $i \in \mathbb{N}$ denotes the *encapsulation level*. To obtain a dcpo, we impose an order on these values, defined by $\mathbf{F}_i \sqsubseteq \mathbf{F}_j \Leftrightarrow i \leq j$, and add an artificial greatest element $\mathbf{F}_\infty$, i.e., $\mathbf{F}_i \sqsubseteq \mathbf{F}_\infty$ for all $i \in \mathbb{N}$. In order to keep our definitions simple, we identify $\perp$ with $\mathbf{F}_0$. Furthermore, we refer to $\mathbf{F}_i$ with $i > 0$ as *finite failure* in the following.

Note that the element $\perp$, usually interpreted as non-termination, is the least element in the domain of failures and, therefore, holds less information than any finite failure. Such an "undefinedness" or "absence of information" element is usually introduced to model lazy or non-strict functions (see also CRWL [12]). Due to our requirement to treat different encapsulation levels in a functional logic computation, we must also treat different levels of "undefinedness" in order to encapsulate the appropriate value sets.[2] Therefore, we also add undefined elements $\mathbf{U}_i$ for all encapsulation levels $i \in \mathbb{N}$ to our semantical domains. These undefined values are only comparable with $\perp$, i.e., $\perp \sqsubseteq \mathbf{U}_i$ for all $i \in \mathbb{N}$.

[2]We present a motivating example for this quite technical issue in Section 5.

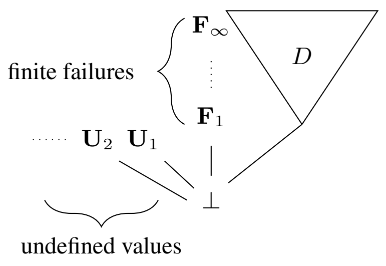

Figure 3: Structure of a dcpo D lifted by $(\cdot)_\mathbf{F}$

As finite failures or undefined values are supposed to be valid inputs for functions and elements in lists, we provide a modified version of the semantics of types that includes all $\mathbf{F}$- and $\mathbf{U}$-values:

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

In this semantics, the new lifting function $(\cdot)_\mathbf{F}$ extends a dcpo with finite failures, undefined values, and a least element. The order on the lifted domain is as described above. Note that finite failures are incomparable to any element of the unlifted dcpo. Figure 3 shows the structure of a dcpo after the lifting. In this modified type semantics, lists can also contain finite failures and different representations of undefined values. Functions are allowed to yield finite failures and different representations of undefined values as results.

4.2 Semantics of Expressions

After adjusting the semantical domains, we are ready to assign denotations to expressions. As in [11], the semantic function $[\![\cdot]\!]^{\tau,n}_\sigma$ is a family of functions where the *environment* σ maps expression variables to semantic values.[3] The index τ indicates the type of the argument of the semantic function. Subsequently, we omit the type index if it is not relevant. The additional index n, not present in [11], identifies different levels of nested encapsulations.

For given σ, τ and n, $[\![\cdot]\!]^{\tau,n}_\sigma$ maps an expression of type τ (determined by type inference) to elements of $\mathcal{P}_H([\![\tau]\!]^\mathbf{F})$. Figure 4 shows the defining equations of TFLC's expression semantics. In contrast to [11], the elements of the Hoare powerdomain may contain finite failures as well as different representations of undefined values and must contain $\perp$, which also entails changes to the semantics of expressions.

As mentioned before, the elements of the Hoare powerdomain are Scott-closed sets. We ensure this property by the down-closure operation $(\cdot){\downarrow}$. The down-closure is defined for every $A \subseteq D$, where $(D, \sqsubseteq)$ is a dcpo, as

$$A{\downarrow} = \{x \in D \mid \exists y \in A.\, x \sqsubseteq y\}$$

For instance, $\{\mathbf{True}\}{\downarrow} = \{\mathbf{True}, \perp\}$, $\{\mathbf{F}_2\}{\downarrow} = \{\mathbf{F}_2, \mathbf{F}_1, \perp\}$ and $\{\mathbf{U}_2\}{\downarrow} = \{\mathbf{U}_2, \perp\}$ w.r.t. the structure shown in Figure 3.

Because $\perp$ is now part of our dcpos, in contrast to the semantics defined in [11], a separate handling of $\sigma(x) = \perp$ is no longer necessary in the case of an expression variable. For the same reason, we now need to down-close the sets on the right-hand side of

[3]By $\sigma[x \mapsto \mathsf{a}]$ we denote the modification of the environment σ where x is mapped to the semantic value a.

$$\llbracket x \rrbracket^i_\sigma = \{\sigma(x)\}{\downarrow} \qquad \llbracket n \rrbracket^i_\sigma = \{n\}{\downarrow}$$

$$\llbracket \mathsf{True} \rrbracket^i_\sigma = \{\mathbf{True}\}{\downarrow} \qquad \llbracket \mathsf{False} \rrbracket^i_\sigma = \{\mathbf{False}\}{\downarrow}$$

$$\llbracket \mathsf{Nil}_\tau \rrbracket^i_\sigma = \{[\,]\}{\downarrow} \qquad \llbracket \mathsf{Cons}(e_1, e_2) \rrbracket^i_\sigma = \bigsqcup_{\mathsf{h} \in \llbracket e_1 \rrbracket^i_\sigma} \bigsqcup_{\mathsf{t} \in \llbracket e_2 \rrbracket^i_\sigma} \{\mathsf{h} : \mathsf{t}\}{\downarrow}$$

$$\llbracket \mathbf{unknown}_\tau \rrbracket^i_\sigma = \llbracket \tau \rrbracket \qquad \llbracket \mathbf{failed}_\tau \rrbracket^i_\sigma = \{\mathbf{F}_i\}{\downarrow}$$

$$\llbracket e_1 + e_2 \rrbracket^i_\sigma = \bigsqcup_{\mathsf{a} \in \llbracket e_1 \rrbracket^i_\sigma} \bigsqcup_{\mathsf{b} \in \llbracket e_2 \rrbracket^i_\sigma} \{\mathsf{a} +^{\mathbf{F}} \mathsf{b}\}{\downarrow}$$

$$\llbracket e_1 \mathbin{?} e_2 \rrbracket^i_\sigma = \llbracket e_1 \rrbracket^i_\sigma \cup \llbracket e_2 \rrbracket^i_\sigma$$

$$\llbracket f \rrbracket^i_\sigma = \{\lambda \mathsf{a}_1. \ldots \{\lambda \mathsf{a}_\mathsf{n}.\ \llbracket e \rrbracket^i_{\sigma[\overline{x_n \mapsto \mathsf{a}_\mathsf{n}}]}\}{\downarrow} \ldots\}{\downarrow} \text{ with } f :: \tau;\ f(\overline{x_n}) = e \text{ in } P$$

$$\llbracket \mathbf{apply}(e_1, e_2) \rrbracket^i_\sigma = \bigsqcup_{\mathsf{f} \in \llbracket e_1 \rrbracket^i_\sigma} \bigsqcup_{\mathsf{a} \in \llbracket e_2 \rrbracket^i_\sigma} (\mathsf{f} \mathbin{\$} \mathsf{a}) \text{ with } \mathsf{f} \mathbin{\$} \mathsf{a} = \begin{cases} \{\mathbf{F}_j\}{\downarrow} & \text{if } \mathsf{f} = \mathbf{F}_j \\ \{\mathbf{U}_j\}{\downarrow} & \text{if } \mathsf{f} = \mathbf{U}_j \\ \mathsf{f}\ \mathsf{a} & \text{otherwise} \end{cases}$$

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

$$\llbracket \mathbf{case}\ e\ \mathbf{of}\ \{\mathsf{True} \to e_1; \mathsf{False} \to e_2\} \rrbracket^i_\sigma = \bigsqcup_{\mathsf{t} \in \llbracket e \rrbracket^i_\sigma} \begin{cases} \{\mathbf{F}_j\}{\downarrow} & \text{if } \mathsf{t} = \mathbf{F}_j \\ \{\mathbf{U}_j\}{\downarrow} & \text{if } \mathsf{t} = \mathbf{U}_j \\ \llbracket e_1 \rrbracket^i_\sigma & \text{if } \mathsf{t} = \mathbf{True} \\ \llbracket e_2 \rrbracket^i_\sigma & \text{if } \mathsf{t} = \mathbf{False} \end{cases}$$

$$\llbracket \mathbf{let}\ x :: \tau = e_1\ \mathbf{in}\ e_2 \rrbracket^i_\sigma = \bigsqcup_{\mathsf{t} \in \llbracket e_1 \rrbracket^i_\sigma} \llbracket e_2 \rrbracket^i_{\sigma[x \mapsto \mathsf{t}]}$$

Figure 4: Adjusted denotational semantics for TFLC expressions

some of the equations. In non-failure cases, only $\bot$ and values including $\bot$ but no finite failures are added to the set (according to the ordering shown in Figure 3). However, if there is a failure $\mathbf{F}_j$ present, then all $\mathbf{F}_i$ with $i < j$ are added. Since we can no longer rely on the property that the base operation $+$ is only applied to non-failure arguments, we use its strict extension $+^{\mathbf{F}}$, which yields the leftmost failure or undefined value, if one of the arguments of $+^{\mathbf{F}}$ is a failure or undefined, and behaves like $+$ otherwise. A free variable, denoted by $\mathbf{unknown}_\tau$, represents all values of its type. Therefore, we can simply use the type semantics defined earlier to specify the semantics of a free variable. Note that we use the version of the type semantics that contains no finite failures, since a free variable that produces all kinds of finite failures would behave quite awkwardly. A finite failure with index i would not necessarily stem from the i-th layer of encapsulation in this case.

Up to now, the index i of the semantic function had no influence on the semantics of expressions (except for $\mathbf{failed}_\tau$ where we use this index to identify the encapsulation level in the failure values). However, the semantics of **allValues** depends on the index. We define it as follows:

$$\llbracket \mathbf{allValues}(e) \rrbracket^{\{\tau\},i}_\sigma = \begin{cases} S{\downarrow} & \text{if } S \subseteq \{\mathbf{F}_i\}{\downarrow} \\ L & \text{if } \mathbf{U}_j \in S \text{ for some } j \\ \{\langle S \rangle, \bot\} & \text{otherwise} \end{cases} \qquad (1)$$

$$\begin{aligned} \text{where}\quad & S = \mathit{nf}_\tau(\llbracket e \rrbracket^{i+1}_{\sigma'}) \\ & L = \{\mathbf{U}_j \mid \mathbf{U}_j \in S, j < i\} \cup \{\bot\} \\ & \sigma'(x) = \mathit{tr}^{\mathbf{U}_i}(\sigma(x)) \text{ for all } x \text{ occurring in } e \end{aligned}$$

The definition requires some explanation: For every dcpo $(D, \sqsubseteq)$ and $A \subseteq D$, $\langle A \rangle$ denotes the set of the compact (i.e., finite) elements in A that are maximal in D (w.r.t. $\sqsubseteq$). The argument to **allValues** is evaluated under an environment σ' obtained by a manipulation of σ. In particular, we restrict it to the variables occurring in the encapsulated expression and replace all occurrences of $\bot$ in the image of σ by $\mathbf{U}_i$. The manipulation allows us to distinguish undefined values coming from the environment from undefined values originally in the encapsulated expression. It can be formalized by the mapping $\mathit{tr}^{\mathbf{U}_i}$ on semantic values defined as follows:

$$\begin{aligned} \mathit{tr}^{\mathbf{U}_i}(\bot) &= \mathbf{U}_i \\ \mathit{tr}^{\mathbf{U}_i}(\mathsf{t}) &= \mathsf{t} && \text{if } \mathsf{t} \in \{\mathbf{True}, \mathbf{False}, [\,]\} \cup \mathbb{N} \\ \mathit{tr}^{\mathbf{U}_i}(\mathsf{t}) &= \mathit{tr}^{\mathbf{U}_i}(\mathsf{t}_1) : \mathit{tr}^{\mathbf{U}_i}(\mathsf{t}_2) && \text{if } \mathsf{t} = \mathsf{t}_1 : \mathsf{t}_2 \end{aligned}$$

Note that this mapping is undefined for functional values, i.e., we do not define the behavior of **allValues** when functional values are passed inside the encapsulated expression.[4] Finally, the normal form function nf represents the complete evaluation of its argument and is defined as the set lifting of nf', where

$$\begin{aligned} \mathit{nf}'_{\mathsf{Bool}}(\mathsf{t}) &= \mathsf{t} \\ \mathit{nf}'_{\mathsf{Nat}}(\mathsf{t}) &= \mathsf{t} \\ \mathit{nf}'_{[\tau]}(\mathsf{t}) &= \begin{cases} \mathit{nf}'_\tau(\mathsf{t}_1) & \text{if } \mathsf{t} = \mathsf{t}_1 : \mathsf{t}_2\ \wedge \mathit{nf}'_\tau(\mathsf{t}_1) \in F \\ \mathit{nf}'_{[\tau]}(\mathsf{t}_2) & \text{if } \mathsf{t} = \mathsf{t}_1 : \mathsf{t}_2\ \wedge \mathit{nf}'_\tau(\mathsf{t}_1) \notin F \\ & \qquad\qquad \wedge\ \mathit{nf}'_{[\tau]}(\mathsf{t}_2) \in F \\ \mathsf{t} & \text{otherwise} \end{cases} \\ & \text{where } F = \{\mathbf{F}_\infty\}{\downarrow} \cup \{\mathbf{U}_i \mid i \in \mathbb{N}\} \end{aligned}$$

Thus, this definition specifies that the first failure which might occur when evaluating nested structures from left to right is returned,

[4]Although the implementation of **allValues** sketched in Section 6 can handle functions, we were not able to define an adequate denotational semantics for it.

similarly to the evaluation of base operations like $+$ (see above).

In order to motivate our definition of the semantics for the primitive $\mathbf{allValues}$ in more detail, let us recall the properties of this encapsulation primitive we are heading for (see Section 3). First of all, we do not want to encapsulate non-determinism that is introduced outside the capsule. This requirement is satisfied since non-deterministic choices outside the capsule lead to different variable bindings and environments σ which are passed to compute the values of the encapsulated expression. Therefore, non-deterministic choices from outside are not visible inside a capsule.

Similarly to the case of non-determinism, failures that are introduced outside a capsule should have no effect on the encapsulation behavior (see Section 3.4). This is ensured since failures are treated like values in the semantics. We use the index i to distinguish failures from the outside from those introduced inside the capsule. This index denotes the nesting level of encapsulation of the current expression. A finite failure $\mathbf{failed}_\tau$ is mapped to the element in the failure domain that corresponds to this level, i.e., the one with the same index (see the definition of the semantics of $\mathbf{failed}_\tau$ in Figure 4). To relate the nesting index i to the actual nesting level, the encapsulated expression e in $\mathbf{allValues}(e)$ is evaluated with an increased index $i+1$, as defined in equation (1) of the definition of $\mathbf{allValues}$. The normal form function nf is used to ensure the requirement for normal form encapsulation (see Section 3.1). It maps a semantic value either to itself, if it is a defined value, or to the leftmost failure or undefined value within the semantic value.

In the defining equation (1), we distinguish three cases:

1. If the resulting set contains only failures from outside the encapsulated expression, i.e., with indices less or equal to i, then the set representing the greatest of these failures is returned.

2. If the resulting set contains some undefined value from the environment, i.e., some $\mathbf{U}_j$, then all undefined values are returned except for those introduced in the current encapsulation. By this mechanism, undefined values of enclosing encapsulations are kept.[5]

3. Otherwise, the set containing the set of all maximal and compact elements of the resulting set is returned.

It should be noted that, since the result of $\mathbf{allValues}$ is intended to be a set, $\mathbf{allValues}$ returns sets as semantic values. To fit these sets into our framework, we need to extend the semantics and the normal form function for the set type:

$$\begin{aligned}
[\![\{\tau\}]\!] &= (\mathcal{P}(\langle[\![\tau]\!]\rangle))_\perp \\
[\![\{\tau\}]\!]^{\mathbf{F}} &= (\mathcal{P}(\langle[\![\tau]\!]\rangle))_{\mathbf{F}} \\
\mathit{nf}'_{\{\tau\}}(\mathsf{t}) &= \mathsf{t}
\end{aligned}$$

To make the domain of sets a dcpo, we impose the discrete order on it. As before, $\perp$ is added as the least element and in the latter type semantics finite failures and undefined values are added.

Finally, we have to specify the semantics for the set operations $\mathbf{size}$, $\mathbf{isEmpty}$, and $\mathbf{minimum}$. Their definitions are straightforward:

$$[\![\mathbf{size}(e)]\!]^i_\sigma = \bigsqcup_{\mathsf{t}\in[\![e]\!]^i_\sigma} \begin{cases} \{\mathbf{F}_j\}{\downarrow} & \text{if } \mathsf{t} = \mathbf{F}_j \\ \{\mathbf{U}_j\}{\downarrow} & \text{if } \mathsf{t} = \mathbf{U}_j \\ \{|\mathsf{t}|\}{\downarrow} & \text{if t is a finite set} \\ \{\perp\} & \text{otherwise} \end{cases}$$

[5]An example showing the need for this special treatment of undefined values is given in Section 5.

$$[\![\mathbf{isEmpty}(e)]\!]^i_\sigma = \bigsqcup_{\mathsf{t}\in[\![e]\!]^i_\sigma} \begin{cases} \{\mathbf{F}_j\}{\downarrow} & \text{if } \mathsf{t} = \mathbf{F}_j \\ \{\mathbf{U}_j\}{\downarrow} & \text{if } \mathsf{t} = \mathbf{U}_j \\ \{\mathbf{True}\}{\downarrow} & \text{if } \mathsf{t} = \emptyset \\ \{\mathbf{False}\}{\downarrow} & \text{otherwise} \end{cases}$$

$$[\![\mathbf{minimum}(e)]\!]^i_\sigma = \bigsqcup_{\mathsf{t}\in[\![e]\!]^i_\sigma} \begin{cases} \{\mathbf{F}_j\}{\downarrow} & \text{if } \mathsf{t} = \mathbf{F}_j \\ \{\mathbf{U}_j\}{\downarrow} & \text{if } \mathsf{t} = \mathbf{U}_j \\ \{\min(\mathsf{t})\}{\downarrow} & \text{if t is a finite set} \\ \{\perp\} & \text{otherwise} \end{cases}$$

Here $|A|$ denotes the cardinality of set A. For "min", False is smaller than True, the ordering on naturals is the usual one and the ordering on lists is the lexicographical ordering. In general an ordering is imposed to user-defined data types by using a lexicographical ordering where constructors that are defined first are always smaller than constructors that are defined later.

Note that $\mathbf{allValues}$ might have an infinite value set as a result. However, only the operation $\mathbf{isEmpty}$ can deliver a result on infinite value sets.

5. EXAMPLES

To demonstrate the application of our semantics and to again highlight the intended behavior of encapsulated search, in this section we calculate the denotational semantics for a number of example programs. Due to [11, Lemma 4.5 – 4.7], it suffices in most cases to calculate with the maximal elements of a Scott-closed set and to treat $\bigsqcup$ as set union. The validity of this property is not obvious for $\mathbf{allValues}$, but, as the examples show, is retained by the transformation $\mathit{tr}^{\mathbf{U}_i}$ performed on values that are passed inside $\mathbf{allValues}$.

The semantics of the expression $\mathbf{allValues}(\mathit{coin})$ can be computed as follows:

$$\begin{aligned}
[\![\mathit{coin}]\!]^i_\emptyset &= [\![0 \;?\; 1]\!]^i_\emptyset = [\![0]\!]^i_\emptyset \cup [\![1]\!]^i_\emptyset \\
&= \{0,\perp\} \cup \{1,\perp\} \\
&= \{0,1,\perp\}
\end{aligned}$$

$$[\![\mathbf{allValues}(\mathit{coin})]\!]^i_\emptyset = \{\langle\{0,1,\perp\}\rangle,\perp\} = \{\{0,1\},\perp\}$$

As intended, the result is the down-closed set that contains the set $\{0,1\}$. The weak encapsulation behavior is demonstrated by the following example from Section 3.2:

$$\begin{aligned}
&[\![\mathbf{let}\ x :: \mathsf{Nat} = \mathit{coin}\ \mathbf{in}\ \mathbf{allValues}(x)]\!]^i_\emptyset \\
&= \bigsqcup_{\mathsf{t}\in[\![\mathit{coin}]\!]^i_\emptyset} [\![\mathbf{allValues}(x)]\!]^i_{[x\mapsto\mathsf{t}]} \\
&= [\![\mathbf{allValues}(x)]\!]^i_{[x\mapsto 0]} \cup [\![\mathbf{allValues}(x)]\!]^i_{[x\mapsto 1]} \\
&\quad \cup [\![\mathbf{allValues}(x)]\!]^i_{[x\mapsto \perp]} \\
&= \{\langle\{0,\perp\}\rangle,\perp\} \cup \{\langle\{1,\perp\}\rangle,\perp\} \cup \{\perp\} \\
&= \{\{0\},\{1\},\perp\}
\end{aligned}$$

The calculation of $[\![\mathbf{allValues}(x)]\!]^i_{[x\mapsto\perp]}$ is the most interesting in the equation above. According to the definition of the semantic function, we need to calculate $\mathit{nf}_{\mathsf{Nat}}([\![x]\!]^{i+1}_{[x\mapsto\mathbf{U}_i]})$ first:

$$\mathit{nf}_{\mathsf{Nat}}([\![x]\!]^{i+1}_{[x\mapsto\mathbf{U}_i]}) = \mathit{nf}_{\mathsf{Nat}}(\{\mathbf{U}_i,\perp\}) = \{\mathbf{U}_i,\perp\}$$

As $\mathbf{U}_i$ is contained in the set that we just calculated, the second rule of the semantics of $\mathbf{allValues}$ is used and, therefore, we get the result $\{\mathbf{U}_j|\mathbf{U}_j \in \{\mathbf{U}_i,\perp\}, j < i\} \cup \{\perp\} = \{\perp\}$.

The next examples show the results of our semantics when failures and non-determinism are combined with encapsulated search.

Without a precise semantics, the results of such expressions are unclear. We consider the following expressions:

$$\mathbf{let}\ x = \mathbf{failed}_{\mathsf{Bool}}\ \mathbf{in}\ \mathbf{allValues}(x\ ?\ \mathsf{True}) \qquad (2)$$
$$\mathbf{let}\ x = \mathbf{failed}_{\mathsf{Bool}}\ \mathbf{in}\ \mathbf{allValues}(x\ ?\ \mathbf{failed}_{\mathsf{Bool}}) \qquad (3)$$

We first calculate the semantics of the common part of both expressions, leaving the argument to **allValues**, named e in the following, unspecified.

$$\begin{aligned}
&[\![\mathbf{let}\ x = \mathbf{failed}_{\mathsf{Bool}}\ \mathbf{in}\ \mathbf{allValues}(e)]\!]^1_{\emptyset}\\
&= \bigsqcup_{\mathsf{t}\in[\![\mathbf{failed}_{\mathsf{Bool}}]\!]^1_{\emptyset}} [\![\mathbf{allValues}(e)]\!]^1_{[x\mapsto\mathsf{t}]}\\
&= \bigsqcup_{\mathsf{t}\in\{\mathbf{F}_1\}\downarrow} [\![\mathbf{allValues}(e)]\!]^1_{[x\mapsto\mathsf{t}]}\\
&= [\![\mathbf{allValues}(e)]\!]^1_{[x\mapsto\mathbf{F}_1]} \cup [\![\mathbf{allValues}(e)]\!]^1_{[x\mapsto\bot]}
\end{aligned}$$

To proceed with the calculation, we substitute e by a concrete expression. To calculate the semantics of expression (2), we choose $e = x\ ?\ \mathsf{True}$:

$$\begin{aligned}
[\![e]\!]^2_{[x\mapsto\mathbf{F}_1]} &= [\![x\ ?\ \mathsf{True}]\!]^2_{[x\mapsto\mathbf{F}_1]}\\
&= [\![x]\!]^2_{[x\mapsto\mathbf{F}_1]} \cup [\![\mathsf{True}]\!]^2_{[x\mapsto\mathbf{F}_1]}\\
&= \{\mathbf{F}_1\}\downarrow \cup \{\mathbf{True}\}\downarrow\\
&= \{\mathbf{F}_1, \mathbf{True}\}\downarrow
\end{aligned}$$

This result is left unchanged by $\mathit{nf}_{\mathsf{Bool}}$. Thus, we meet the third case of the semantic definition of **allValues** and obtain

$$[\![\mathbf{allValues}(e)]\!]^1_{[x\mapsto\mathbf{F}_1]} = \{\langle\{\mathbf{F}_1, \mathbf{True}\}\rangle, \bot\} = \{\{\mathbf{True}\}, \bot\}$$

since $\mathbf{F}_1$ is not maximal w.r.t. our ordering. Similarly, we calculate the semantic value of $[\![\mathbf{allValues}(e)]\!]^1_{[x\mapsto\bot]}$ where we have to change the environment $[x \mapsto \bot]$ to $[x \mapsto \mathbf{U}_1]$ according to the definition of **allValues**:

$$[\![e]\!]^2_{[x\mapsto\mathbf{U}_1]} = \cdots = \{\mathbf{U}_1, \mathbf{True}\}\downarrow$$

Again, the result is left unchanged by $\mathit{nf}_{\mathsf{Bool}}$ so that we meet the second case of the semantic definition of **allValues** and obtain

$$[\![\mathbf{allValues}(e)]\!]^1_{[x\mapsto\mathbf{U}_1]} = \{\bot\}$$

Thus, we get $\{\{\mathbf{True}\}, \bot\}$ as the semantics of the expression (2).

In the same manner, we can calculate the semantics of the expression (3) and substitute e by $x\ ?\ \mathbf{failed}_{\mathsf{Bool}}$: From

$$\begin{aligned}
[\![e]\!]^2_{[x\mapsto\mathbf{F}_1]} &= [\![x\ ?\ \mathbf{failed}_{\mathsf{Bool}}]\!]^2_{[x\mapsto\mathbf{F}_1]}\\
&= [\![x]\!]^2_{[x\mapsto\mathbf{F}_1]} \cup [\![\mathbf{failed}_{\mathsf{Bool}}]\!]^2_{[x\mapsto\mathbf{F}_1]}\\
&= \{\mathbf{F}_1\}\downarrow \cup \{\mathbf{F}_2\}\downarrow\\
&= \{\mathbf{F}_2\}\downarrow
\end{aligned}$$

we obtain $[\![\mathbf{allValues}(e)]\!]^1_{[x\mapsto\mathbf{F}_1]} = \{\langle\{\mathbf{F}_2\}\rangle, \bot\} = \{\{\}, \bot\}$ by the third case of the definition of **allValues**. Similarly, we have

$$[\![e]\!]^2_{[x\mapsto\mathbf{U}_1]} = \cdots = \{\mathbf{U}_1, \mathbf{F}_2\}\downarrow$$

so that we obtain

$$[\![\mathbf{allValues}(e)]\!]^1_{[x\mapsto\mathbf{U}_1]} = \{\bot\}$$

by the second case of the definition of **allValues**. Altogether, we get $\{\{\}, \bot\}$ as the semantics of the expression (3), i.e., the failure inside **allValues** is encapsulated and contributes to an empty failure set. This would not be the case for the expression

$$[\![\mathbf{let}\ x = \mathbf{failed}_{\mathsf{Bool}}\ \mathbf{in}\ \mathbf{allValues}(x)]\!]^1_{\emptyset}$$

which has the semantic value $\{\mathbf{F}_1\}\downarrow$, i.e., it fails instead of returning some value set.

However, if we encapsulate this expression by another outer call to **allValues**, we get the following:

$$[\![\mathbf{allValues}(\mathbf{let}\ x = \mathbf{failed}_{\mathsf{Bool}}\ \mathbf{in}\ \mathbf{allValues}(x))]\!]^1_{\emptyset} = \{\{\}, \bot\}$$

The result is the set with the empty value set as element, since now the failure is introduced inside the outer **allValues** and, thus, the third rule of the semantics of **allValues** is used.

The next example demonstrates the need for the different representations of undefined values in our semantics. Consider the following calculation:

$$\begin{aligned}
&[\![\mathbf{let}\ x = \mathsf{False}\ \mathbf{in}\ \mathbf{allValues}(x\ ?\ \mathsf{True})]\!]^1_{\emptyset}\\
&= \bigsqcup_{\mathsf{t}\in[\![\mathsf{False}]\!]^1_{\emptyset}} [\![\mathbf{allValues}(x\ ?\ \mathsf{True})]\!]^1_{[x\mapsto\mathsf{t}]}\\
&= [\![\mathbf{allValues}(x\ ?\ \mathsf{True})]\!]^1_{[x\mapsto\mathbf{False}]}\\
&\quad \cup [\![\mathbf{allValues}(x\ ?\ \mathsf{True})]\!]^1_{[x\mapsto\bot]}\\
&= \{\{\mathbf{True}, \mathbf{False}\}, \bot\} \cup \{\bot\}
\end{aligned}$$

because we have

$$[\![x\ ?\ \mathsf{True}]\!]^2_{[x\mapsto\mathbf{False}]} = \{\mathbf{True}, \mathbf{False}, \bot\}$$

and

$$[\![x\ ?\ \mathsf{True}]\!]^2_{[x\mapsto\mathbf{U}_1]} = \{\mathbf{U}_1, \mathbf{True}, \bot\}$$

Note that, without our special treatment of encapsulated "unknown" values by introducing $\mathbf{U}_i$ in our semantic definition of **allValues**, we would have obtained the unintended semantics

$$\{\{\mathbf{True}, \mathbf{False}\}, \{\mathbf{True}\}, \bot\}$$

for this expression. This shows the motivation for our domain structure shown in Figure 3. Nevertheless, the construction of other domains with a simpler definition is an interesting topic for future work.

In Section 3.4 we considered the encapsulation of failing computations and argued that the evaluation of $\mathit{consP}(\mathbf{failed}_{[\mathsf{Nat}]})$ should fail. In order to calculate the semantics of $\mathit{consP}(\mathbf{failed}_{[\mathsf{Nat}]})$, we need to calculate the semantics of the function nilP applied to a variable that is bound to a finite failure from a lower level of encapsulation. Note that the finite failure has index 1 and the semantic function is indexed by 2.

$$\begin{aligned}
&[\![\mathit{nilP}(z)]\!]^2_{[z\mapsto\mathbf{F}_1]}\\
&= \bigsqcup_{\mathsf{f}\in[\![\mathit{nilP}]\!]^2_{[z\mapsto\mathbf{F}_1]}} \bigsqcup_{\mathsf{a}\in[\![z]\!]^2_{[z\mapsto\mathbf{F}_1]}} (\mathsf{f}\ \$\ \mathsf{a})\\
&= \lambda\mathsf{a}_1.\left[\!\!\left[\begin{array}{l}\mathbf{case}\ x\ \mathbf{of}\\ \quad\{\mathsf{Nil} \to \mathsf{True}\\ \quad;\mathsf{Cons}(y, ys) \to \mathbf{failed}_{\mathsf{Bool}}\}\end{array}\right]\!\!\right]^2_{[z\mapsto\mathbf{F}_1, x\mapsto\mathsf{a}_1]} \$\ \mathbf{F}_1\\
&= \left[\!\!\left[\begin{array}{l}\mathbf{case}\ x\ \mathbf{of}\ \{\mathsf{Nil} \to \mathsf{True}\\ \qquad\qquad ;\mathsf{Cons}(y, ys) \to \mathbf{failed}_{\mathsf{Bool}}\}\end{array}\right]\!\!\right]^2_{[z\mapsto\mathbf{F}_1, x\mapsto\mathbf{F}_1]}\\
&= \{\mathbf{F}_1, \bot\}
\end{aligned}$$

Since z is bound to a finite failure, the pattern matching in the body of nilP just returns this finite failure.

When we apply our encapsulation primitive **allValues** to the expression $nilP(z)$, the failure is returned but not encapsulated, since the failure is introduced outside the capsule:

$$\llbracket \mathbf{allValues}(nilP(z)) \rrbracket^1_{[z\mapsto \mathbf{F}_1]} = \{\mathbf{F}_1, \bot\}$$

Here the first case of the definition of **allValues** is used.

As we always calculate with down-closed sets, we also need to pass $\bot$ through the variable z. In this case, due to our transformation on the environment, we need to calculate

$$\llbracket nilP(z) \rrbracket^2_{[z\mapsto \mathbf{U}_1]} = \{\mathbf{U}_1, \bot\}$$

and with the second case of the semantics of **allValues** and

$$\{\mathbf{U}_j | \mathbf{U}_j \in \{\mathbf{U}_1, \bot\}, j < 1\} \cup \{\bot\} = \{\bot\}$$

the result of $\llbracket \mathbf{allValues}(nilP(z)) \rrbracket^1_{[z\mapsto \bot]}$ is $\{\bot\}$.

Now we are ready to calculate the semantics of our initial expression:

$$\begin{aligned}
&\llbracket consP(\mathbf{failed}_{[\mathsf{Nat}]}) \rrbracket^1_\emptyset = \bigsqcup_{\mathsf{f}\in\llbracket consP\rrbracket^1_\emptyset} \bigsqcup_{\mathsf{a}\in\llbracket \mathbf{failed}_{[\mathsf{Nat}]}\rrbracket^1_\emptyset} (\mathsf{f}\ \$\ \mathsf{a}) \\
&= \lambda \mathsf{a}_1.\ \llbracket \mathbf{isEmpty}(\mathbf{allValues}(nilP(z)))\rrbracket^1_{[z\mapsto \mathsf{a}_1]}\ \$\ \mathbf{F}_1 \\
&\quad \cup \lambda \mathsf{a}_1.\ \llbracket \mathbf{isEmpty}(\mathbf{allValues}(nilP(z)))\rrbracket^1_{[z\mapsto \mathsf{a}_1]}\ \$\ \bot \\
&= \llbracket \mathbf{isEmpty}(\mathbf{allValues}(nilP(z)))\rrbracket^1_{[z\mapsto \mathbf{F}_1]} \\
&\quad \cup \llbracket \mathbf{isEmpty}(\mathbf{allValues}(nilP(z)))\rrbracket^1_{[z\mapsto \bot]} \\
&= \bigsqcup_{\mathsf{t}\in\llbracket \mathbf{allValues}(nilP(z))\rrbracket^1_{[z\mapsto \mathbf{F}_1]}} \begin{cases} \{\mathbf{F}_j\}{\downarrow} & \text{if } \mathsf{t} = \mathbf{F}_j \\ \{\mathbf{U}_j\}{\downarrow} & \text{if } \mathsf{t} = \mathbf{U}_j \\ \{\mathbf{True}\}{\downarrow} & \text{if } \mathsf{t} = \emptyset \\ \{\mathbf{False}\}{\downarrow} & \text{otherwise} \end{cases} \\
&\quad \cup \bigsqcup_{\mathsf{t}\in\llbracket \mathbf{allValues}(nilP(z))\rrbracket^1_{[z\mapsto \bot]}} \begin{cases} \{\mathbf{F}_j\}{\downarrow} & \text{if } \mathsf{t} = \mathbf{F}_j \\ \{\mathbf{U}_j\}{\downarrow} & \text{if } \mathsf{t} = \mathbf{U}_j \\ \{\mathbf{True}\}{\downarrow} & \text{if } \mathsf{t} = \emptyset \\ \{\mathbf{False}\}{\downarrow} & \text{otherwise} \end{cases} \\
&= \{\mathbf{F}_1\}{\downarrow} \cup \{\bot\} \\
&= \{\mathbf{F}_1, \bot\}
\end{aligned}$$

As mentioned in the introduction, we can also show that particular program transformations used in purely functional languages are no longer valid in the presence of **allValues**. For instance, Peyton Jones et al. [24] argue that, for efficiency reasons, a compiler should be able to change the order of evaluation. In particular, the following two expressions should be interchangeable by a compiler:

```
case x of                       | case y of
  (a,b)  →  case y of           |   (c,d)  →  case x of
              (c,d)  →  e       |               (a,b)  →  e
```

The first expression enforces the evaluation of x before y. In the second expression the pattern matchings are switched and, thus, y is evaluated before x. A similar transformation is proposed in [13] to improve the performance of the Curry system KiCS2 [6]. However, we can show that this transformation is not semantics-preserving if calls to **allValues** occur in the transformed expressions. For this purpose, consider two binary functions g_1 and g_2. Both functions perform pattern matching by means of case expressions on both of their arguments. The only difference is that g_1 evaluates its first argument first while g_2 evaluates its second argument first. Hence, we get

$$\begin{aligned}
\llbracket g_1(x,y)\rrbracket^{i+1}_{[x\mapsto \mathbf{F}_i, y\mapsto \mathbf{F}_{i+1}]} &= \{\mathbf{F}_i\}{\downarrow} \\
\llbracket g_2(x,y)\rrbracket^{i+1}_{[x\mapsto \mathbf{F}_i, y\mapsto \mathbf{F}_{i+1}]} &= \{\mathbf{F}_{i+1}\}{\downarrow}
\end{aligned}$$

Consequently, exchanging g_1 and g_2 in some expression encapsulated by **allValues** might turn a correct result into a failure or vice versa. For example, we have the following semantic value (type annotations and the passing of $\bot$ into **allValues** are omitted):

$$\begin{aligned}
&\llbracket \mathbf{let}\ x = \mathbf{failed}\ \mathbf{in}\ \mathbf{allValues}(g_1(x, \mathbf{failed}))\rrbracket^i_\emptyset \\
&= \bigsqcup_{\mathsf{t}\in\llbracket \mathbf{failed}\rrbracket^i_\emptyset} \llbracket \mathbf{allValues}(g_1(x, \mathbf{failed}))\rrbracket^i_{[x\mapsto \mathsf{t}]} \\
&= \llbracket \mathbf{allValues}(g_1(x, \mathbf{failed}))\rrbracket^i_{[x\mapsto \mathbf{F}_i]} \\
&= \{\mathbf{F}_i\}{\downarrow}
\end{aligned}$$

Here the first case of **allValues** is used since the result of g_1 is the failure passed from outside.

However, replacing g_1 by g_2 yields a different semantic value:

$$\begin{aligned}
&\llbracket \mathbf{let}\ x = \mathbf{failed}\ \mathbf{in}\ \mathbf{allValues}(g_2(x, \mathbf{failed}))\rrbracket^i_\emptyset \\
&= \bigsqcup_{\mathsf{t}\in\llbracket \mathbf{failed}\rrbracket^i_\emptyset} \llbracket \mathbf{allValues}(g_2(x, \mathbf{failed})\rrbracket^i_{[x\mapsto \mathsf{t}]} \\
&= \llbracket \mathbf{allValues}(g_2(x, \mathbf{failed})\rrbracket^i_{[x\mapsto \mathbf{F}_i]} \\
&= \{\{\}, \bot\}
\end{aligned}$$

Here the third case of **allValues** is used, because the result of g_2 is the failure that is introduced inside the argument.

6. IMPLEMENTATION

In the following we sketch an implementation of encapsulated search based on the presented semantics. The implementation is available in the form of set functions in the current release of KiCS2 [16]. For the sake of simplicity, we omit some details.

The Curry implementation KiCS2 translates Curry programs into purely functional Haskell programs. The generated Haskell programs represent the search space, i.e., the non-deterministic results of a computation, as a tree-like data structure. Curry provides an encapsulation operator **getSearchTree**, which returns a representation of this search space. A search strategy can be defined as a tree traversal of this representation. For instance, KiCS2 already provides depth-first, breadth-first, iterative deepening, and parallel search. Due to the demand-driven evaluation strategy, one can also deal with infinite search trees as long as one is interested only in some finite parts of them. A detailed description of the compilation scheme of KiCS2 together with some benchmarks can be found in [6]. In the following, we shortly recapitulate some of the concepts that are necessary to understand the implementation of weak encapsulation according to our semantics.

To represent non-deterministic results in a data structure, KiCS2 extends each data type of the source program by constructors representing a choice between two values and a failure. The latter is necessary since, in functional logic programs, failures are not runtime errors (as in Haskell) but can be constructively used in programs, as shown in Section 3.4. For instance, the Curry data type for the Boolean values (True, False) is translated into the Haskell data type[6]

[6]Actually, the KiCS2 compiler performs some renamings to avoid name conflicts.

```
data Bool = True | False
          | Choice Bool Bool | Fail
```

where `Fail` represents a failure and `(Choice t t′)` a non-deterministic choice between two values `t` and `t′`.[7] For instance, an expression like (True ? False) will be evaluated to

```
(Choice True False)
```

In the following we use the fonts from Section 2 when we refer to Curry syntax (True, *nilP*) and use a different font to refer to Haskell syntax (`True`, `nilP`).

To show the usage of the `Fail` constructor, we consider the operation *nilP* defined in Section 3.4. This operation will be translated into the following Haskell code where the list data type is translated similarly to the Boolean data type:

```
nilP :: List  → Bool
nilP Nil            = True
nilP (Cons y ys)    = Fail
nilP (Choice t t′) = Choice (nilP t) (nilP t′)
nilP Fail           = Fail
```

Hence, `nilP` evaluates to `Fail` if it is applied to a `Cons` or some failure, and a non-deterministic choice in an argument is moved to the result level (this is also called a "pull-tab" step in [2]).

To provide an implementation scheme for **allValues**, we assume an abstract data type `{a}` representing sets with elements of type `a` and we write $\{x_1, \ldots, x_n\}$ to construct a set with elements $x_1, \ldots, x_n$. For the sake of brevity, we use an imaginary pattern `{..}` to perform pattern matching on sets. This pattern matches either an empty or a non-empty set. We start with a naïve definition of **allValues** for Boolean expressions in our target language Haskell that does not distinguish between non-determinism and failures introduced outside or inside the encapsulation primitive. Thus, it implements strong encapsulation. Later we refine its implementation to achieve the behavior proposed in this paper:

```
allValues :: Bool  → {Bool}
allValues True            = {True}
allValues False           = {False}
allValues (Choice t t′) = union (allValues t)
                                (allValues t′)
allValues Fail            = {}
```

The operation `allValues` applied to a Boolean value just returns the singleton set that contains that value. If the argument is a choice, `allValues` is applied to both of its branches and the union of the results is returned. A failing computation has no results, thus, `allValues` applied to the `Fail` constructor yields the empty set.

To achieve weak encapsulation and to be able to handle finite failures as intended, we have to distinguish non-determinism and failures that are introduced outside the primitive **allValues** from those that are introduced inside. Therefore, the `Choice` and the `Fail` constructors both get an extra argument to identify the encapsulation level[8].

For instance, the representation of Booleans in Haskell changes to the following code:

```
data Bool = True | False
          | Choice Int Bool Bool | Fail Int
```

[7]In the actual implementation described in [6], each `Choice` constructor has an additional argument to identify choices stemming from the same subexpression in order to correctly implement call-time choice semantics [19]. Since this aspect is outside the scope of this paper, we omit it here.

[8]The idea of decorating choices to implement weak encapsulation is due to [7]. However, that implementation does not cover failures which leads to unintended results.

Additionally, every function is supplied with an argument that denotes the encapsulation level that is active when the function is applied:

```
nilP :: List  → Int  → Bool
nilP Nil                _   = True
nilP (Cons y ys)       enc = Fail enc
nilP (Choice d t t′) enc = Choice d (nilP t enc)
                                     (nilP t′ enc)
nilP (Fail d)           _   = Fail d
```

When `Fail` and `Choice` constructors are created (as in the second equation), they get the current encapsulation level that is passed to the function via the aditional argument of type `Int`. The levels of already existing `Fail` and `Choice` constructors are retained (as in the third and fourth equation).

To increase the nesting level for non-determinism and failures that are introduced inside an encapsulated expression, we use a source code transformation. It is applied to every subexpression of an encapsulated expression by transforming every source code expression **allValues**(e) into

```
(allValues e′ (enc + 1))
```

where e' is obtained from e by replacing every function application $f(x)$ in e by `f` x `(enc + 1))`, where `enc` is the encapsulation level in which the surrounding context is evaluated. For example, a function declaration

$$f(x) = \textbf{allValues}(g(\textbf{allValues}(h(x)), x))$$

is transformed to

```
f x enc =
  allValues (g (allValues (h x ((enc + 1) + 1))
                         ((enc + 1) + 1))
              (enc + 1))
           (enc + 1)
```

In order to distinguish non-determinism and failures introduced inside from those introduced outside, we refine the implementation of `allValues` in order to check if the encapsulation level is equal to the current encapsulation level (introduced inside) or smaller (introduced outside). Rules that have not changed are omitted.

```
allValues (Choice d t t′) enc
  | d = enc   = union (allValues t enc)
                      (allValues t′ enc)
  | otherwise = Choice d (allValues t enc)
                         (allValues t′ enc)
allValues (Fail d) enc
  | d = enc   = {}
  | otherwise = Fail d
```

If the encapsulation level is equal to the current level, then non-determinism and failures are encapsulated as before. Otherwise, the non-determinism and failures are preserved and the result is a non-deterministic choice between two sets or a failure, respectively.

The above implementation handles non-determinism and finite failures correctly w.r.t. weak encapsulation. To ensure the *completeness of results* property discussed in Section 3.3, we implement the union operation as follows:

```
union (Fail d)    (Fail d′) = Fail (max d d′)
union s@{..}      s′        = union′ s s′
union (Fail d)    s@{..}    = s
union f@(Fail _)  (Choice d t t′)
                            = Choice d (union f t)
                                       (union f t′)
union (Choice d t t′) s     = Choice d (union t s)
                                       (union t′ s)
```

If both arguments are failures, the one with the greater encapsulation level is retained since this one will yield the most defined value. The pattern `s@{..}` matches a—possibly empty—set that is neither a choice nor a failure and binds the set to the variable `s`. If one argument of `union` is such a set, the result must contain all elements of this set. Hence, `union'`, used in the second rule, lazily computes the union of two sets while ignoring any failure in its second argument. `Choice` constructors are propagated to the top.

This definition is closely related to the semantics of **allValues** from Equation 1. The first case of the semantics is used if there are only finite failures that stem from outside. Then the result is the down-closed set of all finite failure values. This set is a representative for the greatest of those failures. In our implementation this behavior is achieved by calculating the maximal failure in the first rule of `union`. The third case of the semantics state that, if there is a non-failure value or a failure that is introduced inside **allValues** in the semantics of the argument, then the result is the possibly empty set containing all these non-failure values. In the implementation, `allValues` creates singleton sets for non-failure values and the empty set for failures that stem from inside. Then the `union` function ignores failures if any of its arguments is a set. The second case of **allValues** and the **U** values have no correspondence in the implementation since those only had to be added for the correct handling of the Scott-closed sets w.r.t. **allValues**.

For the sake of simplicity, we have defined `allValues` for Booleans only. The same scheme can be used for other (non-functional) types. If structured types, like lists, occur, one has to compute the normal form of the encapsulated expression (see Section 3.1) by applying the normal form operator ($!!) [18].

To demonstrate the behavior of our implementation, consider the example from Section 3.4. The function *consP* is transformed to Haskell as follows:

```
consP z enc = isEmpty (allValues (nilP z (enc + 1))
                                 (enc + 1)
                      enc
```

When we apply `consP` to a finite failure, here denoted by `failed`, and an initial encapsulation level `0`, we get the following evaluation:

```
consP (failed 0) 0
   ⇒ isEmpty (allValues (nilP (failed 0) 1) 1) 0
   ⇒ isEmpty (allValues (nilP (Fail 0) 1) 1) 0
   ⇒ isEmpty (allValues (Fail 0) 1) 0
   ⇒ isEmpty (Fail 0) 0
   ⇒ Fail 0
```

As intended, the result is a finite failure rather than the value `True`.

Since the additional argument for the encapsulation level is added to *every* function in the generated Haskell program, the reader might wonder if programs that do not make use of encapsulated search are unnecessarily slowed down. However, our benchmarks that compare implementations with and without the additional argument for a number of such programs show no differences in the execution time.

7. CONCLUSION AND RELATED WORK

In this paper we presented the first formal semantics for a primitive to encapsulate non-deterministic computations in a weak manner. Weak encapsulation is intended to distinguish non-deterministic and failing computations inside and outside the capsule. This distinction is essential to obtain a modular semantics for encapsulation. We analyzed the various alternatives and requirements (normal form encapsulation, completeness of results, weak encapsulation of non-deterministic and failing computations) before we proposed an encapsulation primitive with an appropriate denotational semantics. As an application of our semantics, we showed that a program transformation used in purely functional languages is not semantics preserving in our extended language framework.

An early and quite flexible approach to encapsulated search in a multi-paradigm language has been proposed for Oz [26]. This proposal is based on a primitive operator to evaluate an expression until a non-deterministic step occurs. In this situation, the different expressions are returned so that the programmer can decide how to proceed with the different non-deterministic expressions. Based on this primitive, one can define and implement different search strategies. This proposal also influenced early proposals for encapsulated search in functional logic languages [17]. Unfortunately, the complications due to lazy evaluation and sharing were not realized (and they are also not covered by Oz due to its strict semantics) so that later proposals refined these aspects in various ways [5, 8, 9, 17, 20, 21].

The Curry compiler MCC [22] provides a weak encapsulation primitive, called `findall` [21], but lacks a formal foundation. Moreover, Braßel et al. [8] point out that apparently equal expressions yield different results with this primitive.

An operational semantics for a strong encapsulation operator `getSearchTree` was presented in [8]. Since strong encapsulation is known to depend on the evaluation order in specific cases, the authors proposed to disallow these cases by means of program analysis. In a subsequent work [9], Braßel and Huch presented a simplified operational semantics for `getSearchTree` that features sharing across non-deterministic computations.

Computations with failures in functional logic programs have also been considered in [20]. The authors define the semantics of an operator to check finitely failed computations by an extension of the rewriting calculus CRWL [12]. However, they do not provide an operator to collect all non-failing results as it is presented here.

Antoy and Hanus [5] proposed set functions to express encapsulated search. Set functions can be considered as weak encapsulation that is restricted to encapsulate the body of function definitions. This restriction does not limit their expressiveness but rather eases the comprehension of encapsulation. While the properties of set functions have guided our design of **allValues**, Antoy and Hanus did not specify the semantics of set functions in the presence of finite failures or nested applications.

Braßel [7] presented an implementation idea for set functions. While this idea conforms with the specification of set functions, its treatment of finite failures does not conform with the intended results of the motivating example given by Antoy and Hanus [5]. This motivated the work presented in this paper where we developed a comprehensive specification of weak encapsulation. Our specification builds the foundation of a practical implementation of set functions that is available in the current release of KiCS2 [16].

Acknowledgements.
The authors are grateful to the anonymous referees for helpful comments and suggestions to improve the paper.

8. REFERENCES

[1] Abramsky, S., Jung, A.: Domain theory. In: Handbook of Logic in Computer Science. vol. 3, pp. 1–168. Oxford University Press (1994)

[2] Alqaddoumi, A., Antoy, S., Fischer, S., Reck, F.: The pull-tab transformation. In: Proc. of the Third International Workshop on Graph Computation Models. pp. 127–132. Enschede, The Netherlands (2010)

[3] Antoy, S., Echahed, R., Hanus, M.: A needed narrowing strategy. Journal of the ACM 47(4), 776–822 (2000)

[4] Antoy, S., Hanus, M.: Functional logic programming. Communications of the ACM 53(4), 74–85 (2010)

[5] Antoy, S., Hanus, M.: Set functions for functional logic programming. In: PPDP'09. pp. 73–82 (2009)

[6] Braßel, B., Hanus, M., Peemöller, B., Reck, F.: KiCS2: A new compiler from Curry to Haskell. In: Proc. WFLP 2011. pp. 1–18. Springer LNCS 6816 (2011)

[7] Braßel, B.: Implementing Functional Logic Programs by Translation into Purely Functional Programs. Ph.D. thesis, Christian-Albrechts University of Kiel, Germany (2011)

[8] Braßel, B., Hanus, M., Huch, F.: Encapsulating non-determinism in functional logic computations. Journal of Functional and Logic Programming 2004(6) (2004)

[9] Braßel, B., Huch, F.: On a tighter integration of functional and logic programming. In: Proc. APLAS 2007. pp. 122–138. Springer LNCS 4807 (2007)

[10] Christiansen, J., Seidel, D., Voigtländer, J.: An adequate, denotational, functional-style semantics for Typed FlatCurry. In: WFLP'11. LNCS, vol. 6559, pp. 119–136 (2011)

[11] Christiansen, J., Seidel, D., Voigtländer, J.: An adequate, denotational, functional-style semantics for typed FlatCurry without letrec. Tech. Rep. IAI-TR-2011-1, University of Bonn (Mar 2011), this is a revised version of [10].

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

[13] Hanus, M.: Improving lazy non-deterministic computations by demand analysis. In: ICLP'12. vol. 17, pp. 130–143. Leibniz International Proceedings in Informatics (LIPIcs) (2012)

[14] Hanus, M.: Functional logic programming: From theory to Curry. In: Programming Logics - Essays in Memory of Harald Ganzinger. pp. 123–168. Springer LNCS 7797 (2013)

[15] Hanus, M., Antoy, S., Braßel, B., Engelke, M., Höppner, K., Koj, J., Niederau, P., Sadre, R., Steiner, F.: PAKCS: The Portland Aachen Kiel Curry System. Available at `http://www.informatik.uni-kiel.de/~pakcs/` (2012)

[16] Hanus, M., Peemöller, B., Reck, F.: KiCS2: The Kiel Curry System (Version 2). Available at `http://www-ps.informatik.uni-kiel.de/kics2/` (2013)

[17] Hanus, M., Steiner, F.: Controlling search in declarative programs. In: PLILP/ALP'98. pp. 374–390 (1998)

[18] Hanus (ed.), M.: Curry: An integrated functional logic language (vers. 0.8.3). Available at `http://www.curry-language.org` (2012)

[19] Hussmann, H.: Nondeterministic algebraic specifications and nonconfluent term rewriting. Journal of Logic Programming 12, 237–255 (1992)

[20] López-Fraguas, F., Sánchez-Hernández, J.: A proof theoretic approach to failure in functional logic programming. Theory and Practice of Logic Programming 4(1), 41–74 (2004)

[21] Lux, W.: Implementing encapsulated search for a lazy functional logic language. In: Functional and Logic Programming, pp. 100–113. Springer LNCS 1722 (1999)

[22] Lux, W.: MCC: The Münster Curry Compiler. Available at `http://danae.uni-muenster.de/~lux/curry/` (2007)

[23] Naish, L.: All solutions predicates in Prolog. In: IEEE International Symposium on Logic Programming. pp. 73–77 (July 1985)

[24] Peyton Jones, S., Reid, A., Hoare, T., Marlow, S., Henderson, F.: A semantics for imprecise exceptions. In: PLDI'99. pp. 25–36 (1999)

[25] Peyton Jones, S. (ed.): Haskell 98 Language and Libraries – The Revised Report. Cambridge University Press (2003)

[26] Schulte, C., Smolka, G.: Encapsulated search for higher-order concurrent constraint programming. In: Proc. of the 1994 International Logic Programming Symposium. pp. 505–520. MIT Press (1994)