Free Theorems for Functional Logic Programs

Jan Christiansen
Christian-Albrechts-Universität zu Kiel
Institut für Informatik
Olshausenstraße 40
24098 Kiel
Germany
jac@informatik.uni-kiel.de

Daniel Seidel * Janis Voigtländer
Rheinische Friedrich-Wilhelms-Universität Bonn
Institut für Informatik
Römerstraße 164
53117 Bonn
Germany
{ds,jv}@informatik.uni-bonn.de

Abstract

Type-based reasoning is popular in functional programming. In particular, parametric polymorphism constrains functions in such a way that statements about their behavior can be derived without consulting function definitions. Is the same possible in a strongly, and polymorphically, typed functional *logic* language? This is the question we study in this paper. Logical features like nondeterminism and free variables cause interesting effects, which we examine based on examples and address by identifying appropriate conditions that guarantee standard free theorems or inequational versions thereof to hold. We see this case study as a stepping stone for a general theory, not provided here, involving the definition of a logical relation and other machinery required for parametricity arguments appropriate to functional logic languages.

Categories and Subject Descriptors F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; D.3.3 [*Programming Languages*]: Language Constructs and Features—Polymorphism; D.3.2 [*Programming Languages*]: Language Classifications—Multiparadigm languages; D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.1.6 [*Programming Techniques*]: Logic Programming

General Terms Languages, Theory, Verification

Keywords relational parametricity, Curry, Haskell

1. Introduction

Free theorems (Reynolds 1983; Wadler 1989) have become a success story in reasoning about programs in functional languages like Haskell. Derived just from the types of functions, they are an important source for statements about the semantics of programs, useful for verifying program transformations (Gill et al. 1993; Svenningsson 2002; Johann 2003; Fernandes et al. 2007; Voigtländer 2008a,c), but also in other areas (Voigtländer 2008b, 2009a).

* This author was supported by the DFG under grant VO 1512/1-1.

PLPV'10, January 19, 2010, Madrid, Spain.

Functional logic languages combine functional programming with notions of logic programming (Hanus 2007). They typically provide features like algebraic data types and higher-order abstraction as known from functional languages, and add nondeterminism and narrowing as known from logic languages. For certain applications, this combined paradigm has been very effective (Naylor et al. 2007; Braßel et al. 2008; Christiansen and Fischer 2008; Hanus and Kluß 2009). But it still lacks behind concerning techniques for analyzing and reasoning about programs. For example, while the functional logic language Curry in particular (Hanus 2006) has a very similar type system as Haskell (except for the absence of type classes), there has been no prior work at all on free theorems for that setting. This is where our journey begins. It does not come to an end in this paper.

Should we expect to be able to simply reuse free theorems known from Haskell? Experience says, no. Even in Haskell, free theorems as originally conceived work only for a certain sublanguage. This coverage has been increased through several extensions in recent years (Johann and Voigtländer 2004; Stenger and Voigtländer 2009; Voigtländer 2009b), but there is still a long way to go to deal with the full language. As radical a departure, from the very language foundations, as adding arbitrary nondeterminism is bound to have a severe impact on semantic considerations, including free theorems. Should we expect our investigation to be an interesting endeavor, maybe even fun? Experience says, yes! Studies like the above have led to some surprising discoveries about the interaction of language features in Haskell, and arguably served to improve the general understanding and appreciation of semantic aspects otherwise often swept under the carpet. They have also identified concepts and notions, restrictions on program behavior, that are of interest and use independently of the specific application of free theorems. Gaining comparable insights about the logical features in Curry would be worthwhile in its own right. After all, the semantic understanding and foundation of functional logic languages is still in flow, much less settled than it is currently the case for Haskell. In that sense, investigating how free theorems fare in Curry could also be seen as a means of exploring and highlighting important intricacies of the language that are actually relevant for *any* kind of reasoning about its programs, not just for type-based reasoning.

2. Issues and Aims

Let us be more concrete. One key feature of Curry is nondeterminism provided by the "function"

$$(?) :: \alpha \to \alpha \to \alpha$$

which given two arguments x and y returns either x or y. The existence of "?" already indicates that standard free theorems do not

carry over. Namely, in Haskell a free theorem can be used to establish that there are only two total functions of type $\alpha \to \alpha \to \alpha$, the (curried) projections fst and snd, but in Curry there is this third possibility. We also see that in Curry a "function" actually corresponds to a "set-valued function". That is indeed the semantic intuition we will be using.

Curry programs often look deceptively like Haskell programs. For example, the well-known function *filter* is defined as follows:

$$\begin{array}{l} \mathit{filter} :: (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha] \\ \mathit{filter}\ \_\ [\,] = [\,] \\ \mathit{filter}\ p\ (a : as) = \\ \quad \textbf{if}\ p\ a\ \textbf{then}\ a : (\mathit{filter}\ p\ as)\ \textbf{else}\ \mathit{filter}\ p\ as \end{array}$$

A standard free theorem states that for every choice of p, h, and as, the semantic equivalence

$$(\mathit{filter}\ p\ (\mathit{map}\ h\ as)) \equiv (\mathit{map}\ h\ (\mathit{filter}\ (p \circ h)\ as)) \quad (1)$$

holds, where

$$\begin{array}{l} \mathit{map} :: (\alpha \to \beta) \to [\alpha] \to [\beta] \\ \mathit{map}\ \_\ [\,] = [\,] \\ \mathit{map}\ h\ (a : as) = (h\ a) : (\mathit{map}\ h\ as) \end{array}$$

Given the above definitions, we can prove equivalence (1) in Haskell even without resorting to free theorems, simply by structural induction on as. But that inductive proof does not apply to Curry. In fact, that is one of the complications one faces when analyzing Curry programs: equational reasoning does not really work, or only when exercised with a lot of care and restraint. All the more, it would be nice to have free theorems available as an alternative route to statements like (1). But even that simple equivalence does not hold in Curry. To see evidence, let us do some experiments in a Curry interpreter (here, KiCSi, the interpreter of KiCS (Braßel and Huch 2009)). We evaluate both sides of the supposed equivalence for specific settings of p, h, and as. First,

```
> filter id (map ((\_ -> True) ? (\_ -> False))
  [0])
[True]
More?
```

At this point, the interpreter has produced one result, or solution, namely the one arising from the first alternative $(\lambda\_ \to \mathsf{True})$ in h. The interpreter then asks whether we want to see more. Answering in the positive, by pressing ↵, we obtain

```
[]
More?
```

(arising from the second alternative in h), and finally,

```
No more Solutions
```

Now for the other side,

```
> map ((\_ -> True) ? (\_ -> False))
  (filter (id . ((\_ -> True) ? (\_ -> False)))
  [0])
[True]
More?

[False]
More?

[]
More?

No more Solutions
```

Somewhat surprisingly, we obtain an additional solution that was not present before. Through further experimentation, a certain pattern emerges: solutions of $(\mathit{filter}\ p\ (\mathit{map}\ h\ as))$ are always also solutions of $(\mathit{map}\ h\ (\mathit{filter}\ (p \circ h)\ as))$, but not always the other way round. So maybe a good and valid replacement for equivalence (1) is

$$(\mathit{filter}\ p\ (\mathit{map}\ h\ as)) \sqsubseteq (\mathit{map}\ h\ (\mathit{filter}\ (p \circ h)\ as)) \quad (2)$$

where $\sqsubseteq$ is inclusion on solution sets. For the specific function *filter* it is indeed so, but of course we would prefer to be able to derive such a statement for *every* function $f :: (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha]$ once and for all. Free theorems of an inequational flavor have been studied before (Johann and Voigtländer 2004; Voigtländer and Johann 2007), so there is reason for hope. Also, we can try to revise the faulty equivalence (1) by imposing conditions on its variables, or in other ways. Specifically, inspection of the above failure for $h = (\lambda\_ \to \mathsf{True})\,?\,(\lambda\_ \to \mathsf{False})$ reveals that the difference between $(\mathit{filter}\ p\ (\mathit{map}\ h\ as))$ and $(\mathit{map}\ h\ (\mathit{filter}\ (p \circ h)\ as))$ disappears if h is explicitly shared in the latter:

```
> let h' = (\_ -> True) ? (\_ -> False)
  in map h' (filter (id . h') [0])
[True]
More?

[]
More?

No more Solutions
```

This is an important lesson to learn when dealing with Curry: in contrast to Haskell, inlining may not only affect the efficiency, but also the semantics of programs. Since in general

$$\begin{array}{c} (\textbf{let}\ h' = h\ \textbf{in}\ \mathit{map}\ h'\ (\mathit{filter}\ (p \circ h')\ as)) \\ \sqsubseteq \\ (\mathit{map}\ h\ (\mathit{filter}\ (p \circ h)\ as)) \end{array} \quad (3)$$

(since less sharing of nondeterminism may expand, but never shrink, the solution set), we could hope that instead of equivalence (1) at least

$$\begin{array}{c} (\mathit{filter}\ p\ (\mathit{map}\ h\ as)) \\ \equiv \\ (\textbf{let}\ h' = h\ \textbf{in}\ \mathit{map}\ h'\ (\mathit{filter}\ (p \circ h')\ as)) \end{array} \quad (4)$$

holds. With the previously problematic $h = (\lambda\_ \to \mathsf{True})\,?\,(\lambda\_ \to \mathsf{False})$ it is indeed so, but a seemingly innocent switch to $h = (\lambda\_ \to \mathsf{True}\ ?\ \mathsf{False})$ shatters even this equivalence:

```
> filter id (map (\_ -> True ? False) [0])
[True]
More?

[]
More?

No more Solutions

> let h' = (\_ -> True ? False)
  in map h' (filter (id . h') [0])
[True]
More?

[False]
More?

[]
More?

No more Solutions
```

The best we can still hope for is

$$\begin{array}{c}(\mathit{filter}\ p\ (\mathit{map}\ h\ as)) \\ \sqsubseteq \\ (\mathbf{let}\ h' = h\ \mathbf{in}\ \mathit{map}\ h'\ (\mathit{filter}\ (p \circ h')\ as))\end{array} \tag{5}$$

which would be a stronger statement than (2) due to (3). We do claim that (5) holds for arbitrary p, h, and as, which settles the inequational question (for the specific function *filter* with its standard definition, not for arbitrary functions of its type). We are still interested in discovering conditions on p, h, and/or as which establish equivalence. It should not come as a big surprise that if h is *deterministic*[1], then (4) and even (1) hold. More fascinating, though, would be to find a condition on h more permissive than determinism but guaranteeing at least (4). Naturally, that condition should be able to discriminate between $h = (\lambda\_ \to \mathsf{True})\,?\,(\lambda\_ \to \mathsf{False})$ and $h = (\lambda\_ \to \mathsf{True}\,?\,\mathsf{False})$.

From the above examples alone, it should be apparent that interesting facets of Curry's semantics, and its interaction with type-based reasoning, are waiting to be revealed. Some more teasers:

- We will find that there are other features than nondeterminism à la "?" that can break free theorems in Curry.
- We will find that (for other functions than *filter*, but of its type) the equivalence (1) can break in the direction opposite to (2) .
- We will find that there can be situations where neither $\sqsubseteq$ nor $\sqsupseteq$ holds. Such an incomparability is something absolutely unheard of for Haskell world free theorems, no matter what language extensions were taken into account.

To keep control over all these phenomena, we identify appropriate ways of dealing with them and provide conditions under which something positive can be said: "this equation/inequation becomes valid" instead of "this free theorem breaks". Our investigation here is example-driven in the sense that we mainly perform a case study for one specific function type, rather than developing a general theory of parametricity for functional logic languages, involving the definition of a logical relation and so forth. But despite this restriction to a specific type we are really investigating free theorems. Namely, we do not restrict to a specific *definition* of a function of that type like we have done in the consideration for *filter* above. And we contend that the insights gained from our case study provide the elements needed for developing the general theory. For the moment, we hope to inspire, not to drown in formalization. (That, we can do later.)

3. An Intuitive Explanation of Free Theorems in Haskell

Consider the type signature

$$f :: [\alpha] \to [\alpha]$$

which also serves as motivating example for Wadler (1989). Clearly, any such f takes lists as input and produces lists as output. Also, the type variable α means that f is polymorphic: the function must work for lists over arbitrary element types. As a consequence, the output list can only contain elements from the input list. After all, the function does not know the element type of the lists it operates over, and hence it cannot make up new elements of any concrete type to put into the output.

> So for any input list l (over any element type) the output list $f\ l$ consists solely of elements from l.

Moreover, since f has no access to any global state or other context, f's decisions about which elements from l to propagate to the output list, and in which order and multiplicity, can only be made based on the input list l.[2] The information about l that can be used to make decisions is rather limited as well. In particular, decisions cannot depend on any specifics of the elements of l, because the function is ignorant of the element type, and so is prevented from analyzing list elements in any way. Actually, f can only inspect the *length* of l to decide about its course of action, because that is the only element-independent "information content" of a list.

> So for any pair of lists l and l' of same length (but possibly over different element types), the lists $f\ l$ and $f\ l'$ are formed by making the same position-wise selections of elements from l and l', respectively.

Now consider the map function as given in Section 2 (i.e., with its full definition, not just its type). Clearly, $map\ h$ for any h preserves the length of any input list. So if $l' = map\ h\ l$, then $f\ l$ and $f\ l'$ are of the same length and contain, at each position, position-wise exactly corresponding elements from l and l', respectively. Moreover, any two position-wise corresponding elements, one from l and one from $l' = map\ h\ l$, are related by the latter being the h-image of the former. Therefore, we have that each position of $f\ l'$ contains the h-image of the element at the same position in $f\ l$.

> So for any list l and (type-appropriate) function h, we have
>
> $$(f\ (map\ h\ l)) \equiv (map\ h\ (f\ l))$$

We have reasoned about the behavior of f without consulting f's definition. Instead, we have based our reasoning on considering what the function as constrained by its type can do. This is what Wadler's methodology of free theorems is about.

Here, the essence is as follows. Ignoring possible nontermination and infinite lists for the moment, every input list can be characterized by a natural number n and elements $x_0, \ldots, x_{n-1}$ such that the list is $[x_0, \ldots, x_{n-1}]$. Moreover, each function $f :: [\alpha] \to [\alpha]$ corresponds to a mapping from such combinations $(n, x_0, \ldots, x_{n-1})$ to output lists $[x_{F(0)}, \ldots, x_{F(m-1)}]$, where m is a natural number and F is a mapping from $\{0, \ldots, m-1\}$ to $\{0, \ldots, n-1\}$, and where m and F may depend on n, but not on $x_0, \ldots, x_{n-1}$.[3] Given this information, we can calculate as follows:

$$\begin{aligned}&(f\ (map\ h\ [x_0, \ldots, x_{n-1}])) \\ \equiv\ &(f\ [h\ x_0, \ldots, h\ x_{n-1}]) \\ \equiv\ &[h\ x_{F(0)}, \ldots, h\ x_{F(m-1)}] \\ \equiv\ &(map\ h\ [x_{F(0)}, \ldots, x_{F(m-1)}]) \\ \equiv\ &(map\ h\ (f\ [x_0, \ldots, x_{n-1}]))\end{aligned}$$

Our later reasoning will take place on the same level.

4. Features of Curry to be Wary of

In Section 2 we have already seen *that* nondeterminism (via "?") can break standard free theorems, but not really *why*, and why in specific ways. To understand the impact of nondeterminism in detail, the evaluation strategy used in Curry has to be taken into account. In particular, sharing of variables and the way choices are made for nondeterministic alternatives are essential. There are basically two options: *call-time choice* and *run-time choice*. While the

[1] One way to formalize this notion is to define that a function $h :: \tau_1 \to \tau_2$ is deterministic if for every $x :: \tau_1$, $(\mathbf{let}\ x' = x\ \mathbf{in}\ (h\ x', h\ x'))$ is equivalent to $(\mathbf{let}\ y = h\ x\ \mathbf{in}\ (y, y))$.

[2] The function cannot, for example, consult the user in any way about what to do.

[3] This reasoning corresponds to the "container view" on lists (Prince et al. 2008).

former chooses an alternative before possibly duplicating a nondeterministic entity during the evaluation, the latter allows choices to be made independently for each copy. A simple example explains the difference best. Consider the nondeterministic entity $coin$, which is defined as follows:

$$coin :: \mathsf{Int}$$
$$coin = 0 \,?\, 1$$

For the expression $coin + coin$, possible solutions are 0, 1, and 2. Now consider the function $double$, which is defined as follows:

$$double :: \mathsf{Int} \to \mathsf{Int}$$
$$double\ x = x + x$$

In call-time choice the decision which nondeterministic alternative of $coin$ to take is made only once in the function application $double\ \ coin$. The two instances of $coin$ in the supposed result $coin + coin$ then take the same nondeterministic branch. That is, either both instances are 0 or both are 1. This shows that beta reduction is not always valid in Curry. In fact, the rewrite calculus CRWL (González-Moreno et al. 1999) uses a kind of beta-value reduction to model call-time choice. In run-time choice the decision for $coin$ in the result $coin + coin$ of $double\ \ coin$ is not shared, and hence it can differ for the two instances of $coin$. Therefore, the two strategies of choosing cause different results. While the possible solutions of $double\ coin$ with call-time choice are only 0 and 2, with run-time choice the solution 1 is possible as well. Curry uses call-time choice, because that seems more intuitive from the programmer's point of view.[4] But it does not necessarily ease the job of reasoning about programs. We will have to be very careful about keeping track of sharing, duplication of variables, and places at which nondeterministic choices are made.

Another feature to consider is possible program failure, corresponding to an empty set of solutions. For example, one below yields only a single result, namely 1.

$$one :: \mathsf{Int}$$
$$one = 1 \,?\, \mathbf{failed}$$

Failing computations are used to prune nondeterministic results. For example, when we apply the following function pId to $coin$ the result is 1.

$$pId :: \mathsf{Int} \to \mathsf{Int}$$
$$pId\ 0 = \mathbf{failed}$$
$$pId\ 1 = 1$$

We get the same result if we omit the first rule. Indeed, in implementations Curry is translated into an intermediate language where the resulting functions yield **failed** for all missing patterns. Besides this interaction with nondeterminism, the situation here is quite similar to that in Haskell, where one has the undefined value $\bot$. Indeed, dealing with Curry's **failed** is possible in close analogy to how Haskell free theorems are adapted to the potential presence of $\bot$, and hence will serve as a warm-up in the next section.

Finally, Curry allows the use of *free variables*, again a feature from logic languages. Free variables are introduced by the keyword **free**. During program evaluation a free variable is instantiated in a demand-driven way. That is, if pattern-matching is performed on a free variable, the variable is instantiated by the constructor from the pattern, with fresh free variables as arguments. This evaluation strategy is called *narrowing*. For example, $bools$ below yields all possible lists over True and False, because x is instantiated first to $[\,]$, then to $(b : bs)$ for fresh free variables b and bs, of which then b is instantiated to False and True, while bs is instantiated to $[\,]$ and $(b' : bs')$ for fresh free variables b' and bs', and so forth.

$$bools :: [\mathsf{Bool}]$$
$$bools = nots\ x$$
$$\quad \mathbf{where}\ x\ \mathbf{free}$$
$$nots :: [\mathsf{Bool}] \to [\mathsf{Bool}]$$
$$nots\ [\,] = [\,]$$
$$nots\ (b : bs) = (\neg\ b) : (nots\ bs)$$

It has been proved that a core language for lazy functional logic programming does not need to provide both nondeterministic choice *and* free variables, as each can be expressed by means of the other (Antoy and Hanus 2006). One could thus assume that the addition of free variables does not lead to further effects beyond those already introduced via "?". Surprisingly, though, that is not the case. Things are more complicated, as we will see in Section 5.3.

5. Free Theorems in Curry

Before we can make formal statements, we have to consider what is our notion of program equivalence ("$\equiv$") or refinement ("$\sqsubseteq$") with respect to which we want to formulate free theorems. Equivalence typically means interchangeability in any program context without influencing the result. For this to make sense, the "result" must be something externally observable, not for example something of function type. Refinement in our setting means in terms of solution sets.[5] So we consider two closed expressions exp and exp' to be related as $exp \sqsubseteq exp'$ if and only if for every program context resulting in a base type (such as Bool and Int, without further structure and the possibility of partially defined values) it holds that putting exp' into that context will lead to a set of solutions which is a superset of that obtained by putting exp into that context. For non-closed expressions, "$\sqsubseteq$" holds if it holds for all consistent replacements of unbound variables[6] in exp and exp' with type-appropriate closed expressions. The relation "$\equiv$" is obtained as intersection of "$\sqsubseteq$" and its inverse "$\sqsupseteq$".

That said, we do not fix a particular formal semantics here to be used for evaluating solution sets. Possibilities would be the rewrite calculus CRWL (González-Moreno et al. 1999) or an operational semantics of Albert et al. (2002). In any case, we are going to use only calculation steps that would be valid with each of those semantics as base.

To investigate how free theorems fare in Curry, we consider relevant features separately, one after another. We do that mainly for the type signature

$$f :: [\alpha] \to [\alpha]$$

and its associated standard free theorem that for every h,

$$(f \circ map\ h) \equiv (map\ h \circ f) \tag{6}$$

5.1 The Impact of Potential Program Failure

Even in the absence of nondeterminism the free theorem (6) can break.

Example 1. Consider the following function:

$$f :: [\alpha] \to [\alpha]$$
$$f\ \_ = [\mathbf{failed}]$$

[4] See the discussion by Antoy (2005), but also Riesco and Rodríguez-Hortalá (2010).

[5] In particular, we do not consider the multiplicity in which each solution is produced.

[6] ... not to be confused with *free variables* as introduced by the Curry keyword **free**.

Then:

```
> head (f (map (\_ -> True) [0]))
No more Solutions
```

But:

```
> head (map (\_ -> True) (f [0]))
True
More?

No more Solutions
```

The issue here is that our assertion in Section 3 that for any input list l (over any element type) the output list $f\ l$ contains solely elements from l cannot be upheld. The expression **failed** corresponds to an empty set of solutions. It also subsumes runtime errors and nonterminating computations. And, crucially, it is available at every type. So in fact f *can* make up a new element, independently of a concrete type, to put into the output, if only the trivial element **failed**. The problem then is that h in general may not preserve **failed**, so it can make a difference whether we first map h over a list and then apply an f that introduces **failed**, or whether we first apply such an f and then map h.

The obvious repair is to require that $(h\ \mathbf{failed}) \equiv \mathbf{failed}$. In fact, in Haskell with possible nontermination one would impose a strictness condition on h that says the very same regarding the undefined value $\bot$. However, we can also first establish an inequational free theorem without such a precondition.

Theorem 1 (no nondeterminism, no free variables).
For every $f :: [\alpha] \to [\alpha]$ and every h, we have

$$(f \circ map\ h) \sqsubseteq (map\ h \circ f)$$

Proof. In the presence of **failed**, an input list is characterized by a natural number n, elements $x_0, \ldots, x_{n-1}$ (possibly including **failed**), and an $e \in \{[\,], \mathbf{failed}\}$ such that the list is $x_0 : \cdots : x_{n-1} : e$.[7] Moreover, each function of type $[\alpha] \to [\alpha]$ corresponds to a mapping from such combinations $(n, x_0, \ldots, x_{n-1}, e)$ to output lists $\lfloor x \rfloor_{F(0)} : \cdots : \lfloor x \rfloor_{F(m-1)} : e'$, where m is a natural number, $e' \in \{[\,], \mathbf{failed}\}$, and for every $0 \leqslant i < m$,

$$\lfloor x \rfloor_{F(i)} = \begin{cases} x_j & \text{if } F(i) = j \in \{0, \ldots, n-1\} \\ \mathbf{failed} & \text{if } F(i) = \mathbf{failed} \end{cases}$$

where F is a mapping from $\{0, \ldots, m-1\}$ to $\{0, \ldots, n-1, \mathbf{failed}\}$, and where m, e', and F may depend on n and e, but not on $x_0, \ldots, x_{n-1}$.

Given this information, we can calculate as follows:

$$\begin{aligned} & (f\ (map\ h\ (x_0 : \cdots : x_{n-1} : e))) \\ \equiv\ & (f\ (h\ x_0 : \cdots : h\ x_{n-1} : e)) \\ \equiv\ & (\lfloor h\ x \rfloor_{F(0)} : \cdots : \lfloor h\ x \rfloor_{F(m-1)} : e') \\ \sqsubseteq\ & (h\ \lfloor x \rfloor_{F(0)} : \cdots : h\ \lfloor x \rfloor_{F(m-1)} : e') \\ \equiv\ & (map\ h\ (\lfloor x \rfloor_{F(0)} : \cdots : \lfloor x \rfloor_{F(m-1)} : e')) \\ \equiv\ & (map\ h\ (f\ (x_0 : \cdots : x_{n-1} : e))) \end{aligned}$$

where for every $0 \leqslant i < m$,[8]

$$\lfloor h\ x \rfloor_{F(i)} = \begin{cases} h\ x_j & \text{if } F(i) = j \in \{0, \ldots, n-1\} \\ \mathbf{failed} & \text{if } F(i) = \mathbf{failed} \end{cases}$$

and thus $\lfloor h\ x \rfloor_{F(i)} \sqsubseteq (h\ \lfloor x \rfloor_{F(i)})$ by the obvious $(h\ x_j) \sqsubseteq (h\ x_j)$ and $\mathbf{failed} \sqsubseteq (h\ \mathbf{failed})$.

Note that Example 1 is consistent with the statement of Theorem 1.

Moreover, if we restrict h to preserve **failed**, we can directly replace "$\sqsubseteq$" by "$\equiv$" in the above proof, and thus get the following equational free theorem.

Theorem 2 (no nondeterminism, no free variables).
For every $f :: [\alpha] \to [\alpha]$ and every h with $(h\ \mathbf{failed}) \equiv \mathbf{failed}$, we have

$$(f \circ map\ h) \equiv (map\ h \circ f)$$

5.2 The Additional Impact of Nondeterminism

In the presence of nondeterminism, can we still count on

$$(f \circ map\ h) \equiv (map\ h \circ f)$$

for every $f :: [\alpha] \to [\alpha]$ and h with $(h\ \mathbf{failed}) \equiv \mathbf{failed}$? No, we cannot!

Example 2. Consider the following function:

$f :: [\alpha] \to [\alpha]$
$f\ [\,] = [\,]$
$f\ (x : \_) = [x, x]$

Then:

```
> (\[y,z] -> y==z) (f (map (\x -> x ? not x)
  [True]))
True
More?

True
More?

No more Solutions
```

But:

```
> (\[y,z] -> y==z) (map (\x -> x ? not x) (f
  [True]))
True
More?

False
More?
...
```

Note that $h = (\lambda x \to x\ ?\ \neg\ x)$ used above *does* satisfy $(h\ \mathbf{failed}) \equiv \mathbf{failed}$, but still the supposed equational free theorem breaks. To illustrate what is going on, we can have a look at "evaluations" of $f\ (map\ h\ [\mathsf{True}])$ and $map\ h\ (f\ [\mathsf{True}])$,

[7] Here, and throughout, we ignore the possibility of infinite structures. Taking it into account would not change the results, only complicate the presentation of our reasoning steps a bit.

[8] Note that it does *not* hold that always $\lfloor h\ x \rfloor_{F(i)} = h\ \lfloor x \rfloor_{F(i)}$.

carefully taking call-time choice and sharing into account. We get

$$\begin{aligned}
&(f\ (map\ h\ [\mathsf{True}]))\\
\equiv\ &(f\ [h\ \mathsf{True}])\\
\equiv\ &(\mathbf{let}\ x = h\ \mathsf{True}\ \mathbf{in}\ [x,x])\\
\equiv\ &(\mathbf{let}\ x = \mathsf{True}\ ?\ \mathsf{False}\ \mathbf{in}\ [x,x])\\
\equiv\ &(\mathbf{let}\ x = \mathsf{True}\ \mathbf{in}\ [x,x])\ ?\ (\mathbf{let}\ x = \mathsf{False}\ \mathbf{in}\ [x,x])\\
\equiv\ &[\mathsf{True},\mathsf{True}]\ ?\ [\mathsf{False},\mathsf{False}]
\end{aligned}$$

and

$$\begin{aligned}
&(map\ h\ (f\ [\mathsf{True}]))\\
\equiv\ &(map\ h\ (\mathbf{let}\ x = \mathsf{True}\ \mathbf{in}\ [x,x]))\\
\equiv\ &(\mathbf{let}\ h' = h\ \mathbf{in}\ \mathbf{let}\ x = \mathsf{True}\ \mathbf{in}\ [h'\ x, h'\ x])\\
\equiv\ &(\mathbf{let}\ h' = (\lambda x \to x\ ?\ \neg\ x)\ \mathbf{in}\ \mathbf{let}\ x = \mathsf{True}\ \mathbf{in}\ [h'\ x, h'\ x])\\
\equiv\ &[(\lambda x \to x\ ?\ \neg\ x)\ \mathsf{True}, (\lambda x \to x\ ?\ \neg\ x)\ \mathsf{True}]\\
\equiv\ &[(\mathsf{True}\ ?\ \mathsf{False}), (\mathsf{True}\ ?\ \mathsf{False})]\\
\equiv\ &[\mathsf{True},\mathsf{True}]\ ?\ [\mathsf{True},\mathsf{False}]\ ?\ [\mathsf{False},\mathsf{True}]\ ?\ [\mathsf{False},\mathsf{False}]
\end{aligned}$$

We see that sharing of a whole function application possibly leads to fewer solutions than sharing of the function and its argument separately. This general fact (independent of free theorems, and valid for full Curry) can be formalized as follows.

Proposition 1. *Let exp be an expression (not necessarily closed, but at least not containing x' unbound). For appropriately typed h and x, we have*

$$\begin{array}{c}(\mathbf{let}\ h' = h\ \mathbf{in}\ \mathbf{let}\ y = h'\ x\ \mathbf{in}\ exp)\\ \sqsubseteq\\ (\mathbf{let}\ h' = h\ \mathbf{in}\ \mathbf{let}\ x' = x\ \mathbf{in}\ exp[y \mapsto h'\ x'])\end{array}$$

where $exp[y \mapsto h'\ x']$ is exp with all unbound occurrences of y syntactically replaced by $(h'\ x')$.

Given this insight, and that the solution set of the same side of the free theorem gets potentially smaller as in the case of just **failed** being present (cf. Theorem 1), we can establish an unconditional, inequational free theorem in the presence of **failed** and nondeterminism as follows.

Theorem 3 (no free variables).
For every $f :: [\alpha] \to [\alpha]$ and every h, we have

$$(f \circ map\ h) \sqsubseteq (map\ h \circ f)$$

Proof. In the presence of **failed** and "?", an input list is characterized by a natural number $k \geqslant 1$ and $l_1, \ldots, l_k$ such that the list is $l_1\ ?\ \cdots\ ?\ l_k$ and each l_i is of either of the forms $[x_0, \ldots, x_{n-1}]$ and $x_0 : \cdots : x_{n-1} : \mathbf{failed}$. Since function application distributes over "?", we can restrict ourselves to the case $k = 1$.

A given function of type $[\alpha] \to [\alpha]$ now maps $x_0 : \cdots : x_{n-1} : e$ with $e \in \{[\,], \mathbf{failed}\}$ to $\mathbf{let}\ x'_0 = x_0, \ldots, x'_{n-1} = x_{n-1}\ \mathbf{in}\ l'_1\ ?\ \cdots\ ?\ l'_{k'}$, where $k' \geqslant 1$ and for every $1 \leqslant i \leqslant k'$, we have that l'_i is $\lfloor x' \rfloor_{F_i(0)} : \cdots : \lfloor x' \rfloor_{F_i(m_i-1)} : e'_i$, where m_i is a natural number, $e'_i \in \{[\,], \mathbf{failed}\}$, and for every $0 \leqslant j < m_i$,

$$\lfloor x' \rfloor_{F_i(j)} = \begin{cases} x'_{u_1}\ ?\ \cdots\ ?\ x'_{u_p} & \text{if } F_i(j) = \{u_1, \ldots, u_p\},\ p \geqslant 1\\ \mathbf{failed} & \text{if } F_i(j) = \emptyset \end{cases}$$

where F_i is a mapping from $\{0, \ldots, m_i - 1\}$ to $\mathcal{P}(\{0, \ldots, n-1\})$, and where k', $m_1, \ldots, m_{k'}$, $e'_1, \ldots, e'_{k'}$, and $F_1, \ldots, F_{k'}$ may depend on n and e, but not on $x_0, \ldots, x_{n-1}$. Again we can restrict ourselves to the case $k' = 1$.

We then calculate as follows:

$$\begin{aligned}
&(f\ (map\ h\ (x_0 : \cdots : x_{n-1} : e)))\\
\equiv\ &(\mathbf{let}\ h' = h\ \mathbf{in}\ f\ (h'\ x_0 : \cdots : h'\ x_{n-1} : e)))\\
\equiv\ &(\mathbf{let}\ h' = h\ \mathbf{in}\\
&\quad \mathbf{let}\ y_0 = h'\ x_0, \ldots, y_{n-1} = h'\ x_{n-1}\ \mathbf{in}\\
&\quad\quad \lfloor y \rfloor_{F_1(0)} : \cdots : \lfloor y \rfloor_{F_1(m_1-1)} : e'_1)\\
\sqsubseteq\ &(\mathbf{let}\ h' = h\ \mathbf{in}\\
&\quad \mathbf{let}\ x'_0 = x_0, \ldots, x'_{n-1} = x_{n-1}\ \mathbf{in}\\
&\quad\quad \lfloor h'\ x' \rfloor_{F_1(0)} : \cdots : \lfloor h'\ x' \rfloor_{F_1(m_1-1)} : e'_1)\\
\sqsubseteq\ &(\mathbf{let}\ h' = h\ \mathbf{in}\\
&\quad \mathbf{let}\ x'_0 = x_0, \ldots, x'_{n-1} = x_{n-1}\ \mathbf{in}\\
&\quad\quad h'\ \lfloor x' \rfloor_{F_1(0)} : \cdots : h'\ \lfloor x' \rfloor_{F_1(m_1-1)} : e'_1)\\
\equiv\ &(map\ h\ (\mathbf{let}\ x'_0 = x_0, \ldots, x'_{n-1} = x_{n-1}\ \mathbf{in}\\
&\qquad\qquad \lfloor x' \rfloor_{F_1(0)} : \cdots : \lfloor x' \rfloor_{F_1(m_1-1)} : e'_1))\\
\equiv\ &(map\ h\ (f\ (x_0 : \cdots : x_{n-1} : e)))
\end{aligned}$$

where the notations $\lfloor y \rfloor_{F_1(j)}$ and $\lfloor h'\ x' \rfloor_{F_1(j)}$ should by now be clear. The first "$\sqsubseteq$" is by Proposition 1. The second one is by $\lfloor h'\ x' \rfloor_{F_1(j)} \sqsubseteq (h'\ \lfloor x' \rfloor_{F_1(j)})$, which holds due to $\mathbf{failed} \sqsubseteq (h'\ \mathbf{failed})$ and the distributivity of function application over "?".

As we have seen, the difference between $f\ (map\ h\ [\mathsf{True}])$ and $map\ h\ (f\ [\mathsf{True}])$ in Example 2 is caused by the interplay of two properties of f and h: f introduces sharing and h is nondeterministic. So in order to recover an equational free theorem, we can try to prevent both, or either, of these characteristics by identifying appropriate restrictions. Preventing that f introduces sharing would currently only be possible by inspecting its syntactic definition. However, this would be squarely in contradiction to the spirit of free theorems, by which only f's type should matter. Hence, we instead focus on the potential nondeterminism of h.[9]

We have already motivated in Section 2 that it would be worthwhile to identify a condition that does not fall back to requiring full determinism, but still prevents the worst breaches of free theorems. Here we propose the following restriction.

Definition 1. A function $h :: \tau_1 \to \tau_2$ is *multi-deterministic* if for every $x :: \tau_1$,

$$\begin{array}{c}(\mathbf{let}\ y = h\ x\ \mathbf{in}\ (y,y))\\ \equiv\\ (\mathbf{let}\ h' = h\ \mathbf{in}\ \mathbf{let}\ x' = x\ \mathbf{in}\ (h'\ x', h'\ x'))\end{array}$$

For example, $h = (\lambda x \to x\ ?\ \neg\ x)$ is *not* multi-deterministic, but $h = (id\ ?\ \neg)$ *is*.[10] Note that every deterministic function is multi-deterministic as well.

The restriction just identified indeed leads to an equational version of Proposition 1.

Proposition 2. *The "$\sqsubseteq$" in Proposition 1 can be replaced by "$\equiv$" if h is multi-deterministic.*

As a consequence, we also obtain an equational version of our free theorem in the presence of **failed** and nondeterminism, under appropriate conditions.

[9] It is worth pointing out, though, that the "multi-determinism" restriction going to be imposed on h in Theorems 4, 6, and 8 becomes superfluous if f does not introduce sharing.

[10] Concerning the discussion in Section 2, note that $h = (\lambda\_ \to \mathsf{True})\ ?\ (\lambda\_ \to \mathsf{False})$ *is* multi-deterministic, but $h = (\lambda\_ \to \mathsf{True}\ ?\ \mathsf{False})$ is *not*.

Theorem 4 (no free variables).
For every $f :: [\alpha] \to [\alpha]$ *and every* h *with* $(h\ \mathbf{failed}) \equiv \mathbf{failed}$*, we have that if* h *is multi-deterministic, then*

$$(f \circ map\ h) \equiv (map\ h \circ f)$$

Proof. The proof is similar to the one for Theorem 3, but with both occurrences of "$\sqsubseteq$" in the calculation replaced by "$\equiv$". For the first one, this is possible by Proposition 2, because h is multi-deterministic, and for the second it is possible because h preserves **failed**.

5.3 Adding Free Variables to the Mix

As already hinted at in Section 4, free variables further complicate things. Even the inequational statement from Theorem 3 breaks.

Example 3. Consider the following function:

$$f :: [\alpha] \to [\alpha]$$
$$f\ \_ = [x]\ \mathbf{where}\ x\ \mathbf{free}$$

Then:

```
> not (head (f (map (\_ -> True) [0])))
True
More?

False
More?

No more Solutions
```

But:

```
> not (head (map (\_ -> True) (f [0])))
False
More?

No more Solutions
```

Suddenly, for the first time, we see the left-hand side $(f \circ map\ h)$ producing, in a certain program context, *more* solutions than the right-hand side $(map\ h \circ f)$, in contrast to the

$$(f \circ map\ h) \sqsubseteq (map\ h \circ f)$$

we always had before.[11]

We can understand this behavior as follows. Let

$$\mathbf{unknown} :: \alpha$$
$$\mathbf{unknown} = x\ \mathbf{where}\ x\ \mathbf{free}$$

and replace the definition of f from Example 3 by the following equivalent definition:

$$f :: [\alpha] \to [\alpha]$$
$$f\ \_ = [\mathbf{unknown}]$$

[11] In Section 2 we mentioned that we will find that there are functions of *filter*'s type for which the equivalence (1) can break in the direction opposite to (2). Indeed, a function similar to the above f confirms this claim.

Then with $h = (\lambda\_ \to \mathsf{True})$ we can "evaluate"

$$\begin{aligned}
&(\neg\ (head\ (f\ (map\ h\ [0]))))\\
\equiv\ &(\neg\ (head\ [\mathbf{unknown}]))\\
\equiv\ &(\neg\ \mathbf{unknown})\\
\equiv\ &(\neg\ \mathsf{False})\ ?\ (\neg\ \mathsf{True})\\
\equiv\ &\mathsf{True}\ ?\ \mathsf{False}
\end{aligned}$$

and

$$\begin{aligned}
&(\neg\ (head\ (map\ h\ (f\ [0]))))\\
\equiv\ &(\neg\ (head\ (map\ h\ [\mathbf{unknown}])))\\
\equiv\ &(\neg\ (head\ [h\ \mathbf{unknown}]))\\
\equiv\ &(\neg\ (h\ \mathbf{unknown}))\\
\equiv\ &(\neg\ \mathsf{True})\\
\equiv\ &\mathsf{False}
\end{aligned}$$

The crucial point is that in the first calculation **unknown** is instantiated to both possible constructors of type Bool because the negation function "¬" has both as alternatives in its pattern-matching. In the second calculation, though, **unknown** is discarded by h and no two alternatives arise.

We mentioned in Section 4 that nondeterministic choice and free variables have been found to be interdefinable (Antoy and Hanus 2006). But then no new effects should occur here compared to what happened in Section 5.2. This apparent contradiction can be explained as follows. The "simulation" of free variables through nondeterministic choice works by enumerating all values of a type, and relying on lazy evaluation to ensure the demand-driven instantiation. However, this really depends on type knowledge. For example, in Curry without free variables we could *conceptually* define a wealth of "generators"

$$\mathbf{unknown}_{\mathsf{Bool}} :: \mathsf{Bool}$$
$$\mathbf{unknown}_{\mathsf{Bool}} = \mathsf{False}\ ?\ \mathsf{True}$$
$$\mathbf{unknown}_{\mathsf{Int}} :: \mathsf{Int}$$
$$\mathbf{unknown}_{\mathsf{Int}} = 0\ ?\ 1\ ?\ 2\ ?\ ...$$
$$\mathbf{unknown}_{[\mathsf{Bool}]} :: [\mathsf{Bool}]$$
$$\mathbf{unknown}_{[\mathsf{Bool}]} = [\,]\ ?\ (\mathbf{unknown}_{\mathsf{Bool}} : \mathbf{unknown}_{[\mathsf{Bool}]})$$
$$...$$

but not a single

$$\mathbf{unknown} :: \alpha$$

with the intended semantics. So this availability of a second entity, beside **failed**, that is available at every type is indeed something new, and must be considered separately from "just" nondeterministic choice. The analogy with **failed** seems to be the key to recovering free theorems in the presence of free variables. Like we always have $\mathbf{failed} \sqsubseteq (h\ \mathbf{failed})$ but must explicitly require the converse if we want equivalence, we also always have $\mathbf{unknown} \sqsupseteq (h\ \mathbf{unknown})$ but must explicitly require equivalence if so desired.

But is it really that simple? Can the treatment of **unknown** be dual to that of **failed**? Surprisingly, no! If it were, we would have

$$(f \circ map\ h) \sqsubseteq (map\ h \circ f)$$

for every $f :: [\alpha] \to [\alpha]$ and every h with $(h\ \mathbf{unknown}) \equiv \mathbf{unknown}$, by adaptation of Theorem 3. But it turns out not to be so.

Example 4. Consider the following function:

$$f :: [\alpha] \to [\alpha]$$
$$f\ [x] = [\mathbf{unknown}, x]$$

Then:

```
> (\[x,y] -> (not x) && y) (f (map (id ? (\_ ->
  True)) [False]))
False
More?

True
More?

False
More?

No more Solutions
```

But:

```
> (\[x,y] -> (not x) && y) (map (id ? (\_ ->
  True)) (f [False]))
False
More?

False
More?

False
More?

No more Solutions
```

The supposed theorem breaks, but why? Obviously, $h = (id\ ?\ (\lambda\_ \to \mathsf{True}))$ *does* preserve $\mathbf{unknown}$, so we might expect

$$(f \circ map\ h) \sqsubseteq (map\ h \circ f)$$

However, not *every alternative* of h preserves $\mathbf{unknown}$. Indeed, $(\lambda\_ \to \mathsf{True})\ \mathbf{unknown} \not\equiv \mathbf{unknown}$.

What we see here is a departure from the supposed duality of $\mathbf{failed}$ and $\mathbf{unknown}$. For any $h = (h_1\ ?\cdots ?\ h_n)$ it clearly holds that if h preserves $\mathbf{failed}$, then each h_i does. In fact, this observation is implicitly used in the proof of Theorem 4. But it is *not* the case that if an h of the given form preserves $\mathbf{unknown}$, then each h_i does. To regain an inequational statement like that from Theorem 3, we have to specify that each alternative of h preserves $\mathbf{unknown}$. However, we want a semantic formulation, not a syntactic formulation relying on a specific form of h. We propose the following characterization.

Definition 2. A function h is *multi-onto* if

$$(\mathbf{let}\ h' = h\ \mathbf{in}\ (h'\ \mathbf{unknown}, h')) \equiv (\mathbf{unknown}, h)$$

For example, $h = (id\ ?\ (\lambda\_ \to \mathsf{True}))$ is *not* multi-onto, because

$$\begin{aligned}
&(\mathbf{let}\ h' = h\ \mathbf{in}\ (h'\ \mathbf{unknown}, h')) :: (\mathsf{Bool}, \mathsf{Bool} \to \mathsf{Bool})\\
\equiv\ &(\mathbf{let}\ h' = id\ \mathbf{in}\ (h'\ \mathbf{unknown}, h'))\\
&\ ?\ (\mathbf{let}\ h' = (\lambda_ \to \mathsf{True})\ \mathbf{in}\ (h'\ \mathbf{unknown}, h'))\\
\equiv\ &(\mathbf{unknown}, id)\ ?\ (\mathsf{True}, (\lambda_ \to \mathsf{True}))\\
\equiv\ &((\mathsf{False}\ ?\ \mathsf{True}), id)\ ?\ (\mathsf{True}, (\lambda_ \to \mathsf{True}))\\
\equiv\ &(\mathsf{False}, id)\ ?\ (\mathsf{True}, id)\ ?\ (\mathsf{True}, (\lambda_ \to \mathsf{True}))
\end{aligned}$$

but

$$\begin{aligned}
&(\mathbf{unknown}, h) :: (\mathsf{Bool}, \mathsf{Bool} \to \mathsf{Bool})\\
\equiv\ &((\mathsf{False}\ ?\ \mathsf{True}), (id\ ?\ (\lambda_ \to \mathsf{True})))\\
\equiv\ &(\mathsf{False}, id)\ ?\ (\mathsf{False}, (\lambda_ \to \mathsf{True}))\ ?\ (\mathsf{True}, id)\\
&\ ?\ (\mathsf{True}, (\lambda_ \to \mathsf{True}))
\end{aligned}$$

On the other hand, $h = (id\ ?\ \neg)$ *is* multi-onto.

The restriction just identified allows us to state the following proposition.

Proposition 3. *Let exp be an expression. For appropriately typed h that is multi-onto, we have*

$$\begin{aligned}
&(\mathbf{let}\ h' = h\ \mathbf{in}\ (\mathbf{unknown}, exp))\\
\equiv\\
&(\mathbf{let}\ h' = h\ \mathbf{in}\ (h'\ \mathbf{unknown}, exp))
\end{aligned}$$

Indeed, our definition of multi-onto captures *exactly* all functions h for which the mentioned equivalence holds for every choice of exp. (Note that exp can of course contain h'.)

Now we can establish a variant of Theorem 3 even for Curry programs with free variables.

Theorem 5 (Curry).
For every $f :: [\alpha] \to [\alpha]$ and every h, we have that if h is multi-onto, then

$$(f \circ map\ h) \sqsubseteq (map\ h \circ f)$$

Proof. The proof is similar to the one for Theorem 3. The F_i now map from $\{0, \ldots, m_i - 1\}$ to $\mathcal{P}(\{0, \ldots, n-1\}) \cup \{\mathbf{unknown}\}$ and we set

$$\lfloor x' \rfloor_{F_i(j)} = \begin{cases} x'_{u_1}\ ?\cdots ?\ x'_{u_p} & \text{if } F_i(j) = \{u_1, \ldots, u_p\},\ p \geqslant 1\\ \mathbf{failed} & \text{if } F_i(j) = \emptyset\\ \mathbf{unknown} & \text{if } F_i(j) = \mathbf{unknown} \end{cases}$$

and analogously for $\lfloor y \rfloor_{F_i(j)}$ and $\lfloor h'\ x' \rfloor_{F_i(j)}$. We then proceed as in the proof of Theorem 3, the crucial point being that in the calculation we still have

$$\begin{aligned}
&(\mathbf{let}\ h' = h\ \mathbf{in}\\
&\quad \mathbf{let}\ x'_0 = x_0, \ldots, x'_{n-1} = x_{n-1}\ \mathbf{in}\\
&\qquad \lfloor h'\ x' \rfloor_{F_1(0)} : \cdots : \lfloor h'\ x' \rfloor_{F_1(m_1-1)} : e'_1)\\
\sqsubseteq\ &(\mathbf{let}\ h' = h\ \mathbf{in}\\
&\quad \mathbf{let}\ x'_0 = x_0, \ldots, x'_{n-1} = x_{n-1}\ \mathbf{in}\\
&\qquad h'\ \lfloor x' \rfloor_{F_1(0)} : \cdots : h'\ \lfloor x' \rfloor_{F_1(m_1-1)} : e'_1)
\end{aligned}$$

thanks to, in addition to the reasons given in the proof of Theorem 3, also Proposition 3.

Without using multi-onto, we get an inequational variant of Theorem 4, valid even for Curry programs with free variables. (And there are counterexamples showing that equivalence cannot be recovered without also imposing that h is multi-onto.)

Theorem 6 (Curry).
For every $f :: [\alpha] \to [\alpha]$ and every h with $(h\ \mathbf{failed}) \equiv \mathbf{failed}$, we have that if h is multi-deterministic, then

$$(f \circ map\ h) \sqsupseteq (map\ h \circ f)$$

Proof. The proof is again similar to the one for Theorem 3. With F_i, $\lfloor x' \rfloor_{F_i(j)}$, $\lfloor y \rfloor_{F_i(j)}$, and $\lfloor h'\ x' \rfloor_{F_i(j)}$ as in the proof of Theo-

rem 5, the crucial calculation steps are as follows:

$$
\begin{aligned}
&(\mathbf{let}\ h' = h\ \mathbf{in} \\
&\quad \mathbf{let}\ y_0 = h'\ x_0, \ldots, y_{n-1} = h'\ x_{n-1}\ \mathbf{in} \\
&\qquad \lfloor y \rfloor_{F_1(0)} : \cdots : \lfloor y \rfloor_{F_1(m_1-1)} : e_1') \\
\equiv\ &(\mathbf{let}\ h' = h\ \mathbf{in} \\
&\quad \mathbf{let}\ x_0' = x_0, \ldots, x_{n-1}' = x_{n-1}\ \mathbf{in} \\
&\qquad \lfloor h'\ x' \rfloor_{F_1(0)} : \cdots : \lfloor h'\ x' \rfloor_{F_1(m_1-1)} : e_1') \\
\sqsupseteq\ &(\mathbf{let}\ h' = h\ \mathbf{in} \\
&\quad \mathbf{let}\ x_0' = x_0, \ldots, x_{n-1}' = x_{n-1}\ \mathbf{in} \\
&\qquad h'\ \lfloor x' \rfloor_{F_1(0)} : \cdots : h'\ \lfloor x' \rfloor_{F_1(m_1-1)} : e_1')
\end{aligned}
$$

where the "$\equiv$" is by Proposition 2 and the "$\sqsupseteq$" is by $\lfloor h'\ x' \rfloor_{F_1(j)} \sqsupseteq (h'\ \lfloor x' \rfloor_{F_1(j)})$, which holds by the distributivity of function application over "?", by $\mathbf{unknown} \sqsupseteq (h'\ \mathbf{unknown})$, and by $\mathbf{failed} \equiv (h'\ \mathbf{failed})$, where we know the latter since h preserves $\mathbf{failed}$.

Of course, Theorems 5 and 6 can be combined to obtain conditions for an equational free theorem $(f \circ map\ h) \equiv (map\ h \circ f)$.

Somewhat obscurely, we can now even have a situation where neither the left-hand side $(f \circ map\ h)$ refines the right-hand side $(map\ h \circ f)$, nor the other way round.

Example 5. Consider the following function:

$f :: [\alpha] \to [\alpha]$
$f\ \_ = [\mathbf{unknown}, \mathbf{failed}]$

Then:

```
> not (head (f (map (\_ -> True) [0])))
True
More?

False
More?

No more Solutions
```

But:

```
> not (head (map (\_ -> True) (f [0])))
False
More?

No more Solutions
```

And:

```
> head (tail (f (map (\_ -> True) [0])))
No more Solutions
```

But:

```
> head (tail (map (\_ -> True) (f [0])))
True
More?

No more Solutions
```

Depending on the program context, the left-hand side produces once strictly more, and once strictly less solutions. Of course, the same is possible for functions of *filter*'s type.

Writing a function such that in some program context the left-hand side and the right-hand side each produce exactly one solution, but different ones, is left as an exercise for the reader. If the reader says "I don't want my free theorems to allow this", then the conditions to prevent it can be found in Theorems 5 and 6.

To wrap up, we give some results that are obtained via the same kind of reasoning found in this section so far, but for the example discussed in Section 2, in the general form of arbitrary functions of *filter*'s type.

Theorem 7 (Curry).
For every $f :: (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha]$ *and every* h*, we have that if* h *is multi-onto, then*

$$(f\ p \circ map\ h) \sqsubseteq (\mathbf{let}\ h' = h\ \mathbf{in}\ map\ h' \circ f\ (p \circ h'))$$

Theorem 8 (Curry).
For every $f :: (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha]$ *and every* h *with* $(h\ \mathbf{failed}) \equiv \mathbf{failed}$*, we have that if* h *is multi-deterministic, then*

$$(f\ p \circ map\ h) \sqsupseteq (\mathbf{let}\ h' = h\ \mathbf{in}\ map\ h' \circ f\ (p \circ h'))$$

Finally, we answer an open question from Section 2, for the specific function *filter*. Since that function does not involve $\mathbf{failed}$ or $\mathbf{unknown}$, we have that for every p and h, if h is multi-deterministic, then

$$(filter\ p \circ map\ h) \equiv (\mathbf{let}\ h' = h\ \mathbf{in}\ map\ h' \circ filter\ (p \circ h'))$$

The equivalence cannot be upheld for h that are not multi-deterministic, because *filter* introduces sharing. In fact, every element ending up in the output list is used twice: first to test the predicate on it, and then to actually put it into the output list.

6. Conclusion

Functional logic programming is receiving increased attention recently (Fischer et al. 2009). We hope to nurture this trend by developing reasoning techniques for languages like Curry. Naturally, techniques developed for functional languages are good candidates for a transfer to functional *logic* languages, but we have seen that interesting new effects can occur.

Free theorems, with their proven usefulness in reasoning about functional programs, are particularly attractive for functional logic programs, because "by-hand" reasoning can be quite expensive in the combined paradigm. For example, one has to be very careful about the interplay between sharing and nondeterminism. Such issues also affect free theorems, but there we can hope to deal with them once and for all in a general type-driven methodology, rather than again and again doing cumbersome proofs for specific functions. As we have seen, proper treatment of sharing is important already in formulating free theorems now. Then, we get free theorems for a subset of Curry which is a proper superset of Curry's functional subset. In particular, we find:

- It is not necessary to require functions to be fully deterministic, which would probably be the first shot. Instead, functions like h in (4) can still contain some nondeterminism without immediately breaking equational free theorems. We introduced *multi-deterministic* functions as an appropriate characterization for what degree of nondeterminism can be tolerated.
- Together with *failure preservation*, which corresponds to strict functions in Haskell, multi-determinism guarantees at least one inequational direction of free theorems to be valid.
- To guarantee the other direction to be valid, even in the presence of free variables, the criterion *multi-onto* is required.

One obvious direction for future work is to develop a full theory of relational parametricity that abstracts from the example types we have investigated. We are convinced that the three points made above will prove true in general, and that our insights gained here will inform the construction of an appropriate logical relation.

Of course, it would be desirable to investigate means of establishing that a given function is multi-deterministic and/or multi-onto. We have on purpose devised the most general semantic characterizations, but in practice static checks will be needed. Also, we mentioned that the multi-determinism requirement can be dropped if we know that the function for which we derive a free theorem does not introduce sharing.[12] A tractable treatment of "does not introduce sharing" could be based on a refined type system, maybe in the spirit of Clean's uniqueness types (Barendsen and Smetsers 1995).

Acknowledgments

We would like to thank Bernd Braßel, Sebastian Fischer, Michael Hanus, Frank Huch, and Fabian Reck for interesting discussions and for comments on a draft of this paper.

References

E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. Operational semantics for functional logic languages. In *Electronic Notes in Theoretical Computer Science*, volume 76. Elsevier, 2002. doi: 10.1016/S1571-0661(04)80782-5.

S. Antoy. Evaluation strategies for functional logic programming. *Journal of Symbolic Computation*, 40(1):875–903, 2005. doi: 10.1016/j.jsc.2004.12.007.

S. Antoy and M. Hanus. Overlapping rules and logic variables in functional logic programs. In *International Conference on Logic Programming, Proceedings*, volume 4079 of *LNCS*, pages 87–101. Springer-Verlag, 2006. doi: 10.1007/11799573_9.

E. Barendsen and S. Smetsers. Uniqueness type inference. In *Programming Languages: Implementations, Logics and Programs, Proceedings*, volume 982 of *LNCS*, pages 189–206. Springer-Verlag, 1995. doi: 10.1007/BFb0026821.

B. Braßel and F. Huch. The Kiel Curry system KiCS. In *International Conference Applications of Declarative Programming and Knowledge Management, INAP 2007, and Workshop on Logic Programming, WLP 2007, Revised Selected Papers*, pages 195–205. Springer-Verlag, 2009. doi: 10.1007/978-3-642-00675-3_13.

B. Braßel, M. Hanus, and M. Müller. High-level database programming in Curry. In *Practical Aspects of Declarative Languages, Proceedings*, volume 4902 of *LNCS*, pages 316–332. Springer-Verlag, 2008. doi: 10.1007/978-3-540-77442-6_21.

J. Christiansen and S. Fischer. EasyCheck: Test data for free. In *Functional and Logic Programming, Proceedings*, volume 4989 of *LNCS*, pages 322–336. Springer-Verlag, 2008. doi: 10.1007/978-3-540-78969-7_23.

J.P. Fernandes, A. Pardo, and J. Saraiva. A shortcut fusion rule for circular program calculation. In *Haskell Workshop, Proceedings*, pages 95–106. ACM Press, 2007. doi: 10.1145/1291201.1291216.

S. Fischer, O. Kiselyov, and C. Shan. Purely functional lazy nondeterministic programming. In *International Conference on Functional Programming, Proceedings*, pages 11–22. ACM Press, 2009. doi: 10.1145/1596550.1596556.

A. Gill, J. Launchbury, and S.L. Peyton Jones. A short cut to deforestation. In *Functional Programming Languages and Computer Architecture, Proceedings*, pages 223–232. ACM Press, 1993. doi: 10.1145/165180.165214.

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. doi: 10.1016/S0743-1066(98)10029-8.

M. Hanus, editor. *Curry: An Integrated Functional Logic Language (Vers. 0.8.2)*. 2006. Available at `http://curry-language.org`.

M. Hanus. Multi-paradigm declarative languages. In *International Conference on Logic Programming, Proceedings*, volume 4670 of *LNCS*, pages 45–75. Springer-Verlag, 2007. doi: 10.1007/978-3-540-74610-2_5.

M. Hanus and C. Kluß. Declarative programming of user interfaces. In *Practical Aspects of Declarative Languages, Proceedings*, volume 5418 of *LNCS*, pages 16–30. Springer-Verlag, 2009. doi: 10.1007/978-3-540-92995-6_2.

P. Johann. Short cut fusion is correct. *Journal of Functional Programming*, 13(4):797–814, 2003. doi: 10.1017/S0956796802004409.

P. Johann and J. Voigtländer. Free theorems in the presence of seq. In *Principles of Programming Languages, Proceedings*, pages 99–110. ACM Press, 2004. doi: 10.1145/982962.964010.

M. Naylor, E. Axelsson, and C. Runciman. A functional-logic library for Wired. In *Haskell Workshop, Proceedings*, pages 37–48. ACM Press, 2007. doi: 10.1145/1291201.1291207.

R. Prince, N. Ghani, and C. McBride. Proving properties about lists using containers. In *Functional and Logic Programming, Proceedings*, volume 4989 of *LNCS*, pages 97–112. Springer-Verlag, 2008. doi: 10.1007/978-3-540-78969-7_9.

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

A. Riesco and J. Rodríguez-Hortalá. Programming with singular and plural non-deterministic functions. In *Partial Evaluation and Program Manipulation, Proceedings*. ACM Press, 2010.

F. Stenger and J. Voigtländer. Parametricity for Haskell with imprecise error semantics. In *Typed Lambda Calculi and Applications, Proceedings*, volume 5608 of *LNCS*, pages 294–308. Springer-Verlag, 2009. doi: 10.1007/978-3-642-02273-9.

J. Svenningsson. Shortcut fusion for accumulating parameters & zip-like functions. In *International Conference on Functional Programming, Proceedings*, pages 124–132. ACM Press, 2002. doi: 10.1145/583852.581491.

J. Voigtländer. Proving correctness via free theorems: The case of the destroy/build-rule. In *Partial Evaluation and Program Manipulation, Proceedings*, pages 13–20. ACM Press, 2008a. doi: 10.1145/1328408.1328412.

J. Voigtländer. Much ado about two: A pearl on parallel prefix computation. In *Principles of Programming Languages, Proceedings*, pages 29–35. ACM Press, 2008b. doi: 10.1145/1328897.1328445.

J. Voigtländer. Semantics and pragmatics of new shortcut fusion rules. In *Functional and Logic Programming, Proceedings*, volume 4989 of *LNCS*, pages 163–179. Springer-Verlag, 2008c. doi: 10.1007/978-3-540-78969-7_13.

J. Voigtländer. Bidirectionalization for free! In *Principles of Programming Languages, Proceedings*, pages 165–176. ACM Press, 2009a. doi: 10.1145/1480881.1480904.

J. Voigtländer. Free theorems involving type constructor classes. In *International Conference on Functional Programming, Proceedings*, pages 173–184. ACM Press, 2009b. doi: 10.1145/1596550.1596577.

J. Voigtländer and P. Johann. Selective strictness and parametricity in structural operational semantics, inequationally. *Theoretical Computer Science*, 388(1–3):290–318, 2007. doi: 10.1016/j.tcs.2007.09.014.

P. Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture, Proceedings*, pages 347–359. ACM Press, 1989. doi: 10.1145/99370.99404.

[12] Interestingly, we cannot trade the multi-onto requirement for absence of sharing.