Formalising FinFuns – Generating Code for Functions as Data from Isabelle/HOL

Andreas Lochbihler

Universität Karlsruhe (TH), Germany
lochbihl@ipd.info.uni-karlsruhe.de

Abstract. FinFuns are total functions that are constant except for a finite set of points, i.e. a generalisation of finite maps. We formalise them in Isabelle/HOL and present how to safely set up Isabelle's code generator such that operations like equality testing and quantification on FinFuns become executable. On the code output level, FinFuns are explicitly represented by constant functions and pointwise updates, similarly to associative lists. Inside the logic, they behave like ordinary functions with extensionality. Via the update/constant pattern, a recursion combinator and an induction rule for FinFuns allow for defining and reasoning about operators on FinFuns that directly become executable. We apply the approach to an executable formalisation of sets and use it for the semantics for a subset of concurrent Java.

1 Introduction

In recent years, executable formalisations, proofs by reflection [8] and automated generators for counter examples [1,5] have received much interest in the theorem proving community. All major state-of-the-art theorem provers like Coq, ACL2, PVS, HOL4 and Isabelle feature some interface to a standard (usually external) functional programming language to directly extract high-assurance code from theorems or proofs or both. Isabelle/HOL provides two code generators [3,6], which support datatypes and recursively defined functions, where Haftmann's [6] is supposed to replace Berghofer's [3]. Berghofer's, which is used to search for counter examples by default (*quickcheck*) [1], can also deal with inductively defined predicates, but not with type classes. Haftmann's additionally supports type classes and output in SML, OCaml and Haskell, but inductively defined predicates are not yet available and *quickcheck* is still experimental.

Beyond these areas, code generation is currently rather limited in Isabelle/HOL. Consequently, the everyday Isabelle user invokes the *quickcheck* facility on some conjecture and frequently encounters an error message such as "`Unable to generate code for` $op = (\lambda x.\ True)$" or "`No such mode [1, 2] for ...`". Typically, such a message means that an assumption or conclusion involves a test on function equality (which underlies both universal and existential quantifiers) or an inductive predicate no code for which can be produced. In particular, the following restrictions curb *quickcheck*'s usefulness:

- Equality on functions is only possible if the domain is finite and enumerable.
- Quantifiers are only executable if they are bounded by a finite set (e.g. $\forall x \in A.\ P\ x$).
- (Finite) sets are explicitly represented by lists, but as the set type has been merged with predicates in version Isabelle 2008, only Berghofer's code generator can work with sets properly.

The very same problems reoccur when provably correct code from a formalisation is to be extracted, although one is willing to commit more effort in adjusting the formalisation and setting up the code generator for it in that case. To apply *quickcheck* to their formalisations, end-users expect to supply little or no effort.

In the area of programming languages, states (like memories, stores, and thread pools) are usually finite, even though the identifiers (addresses, variable names, thread IDs, ...) are typically taken from an infinite pool. Such a state is most easily formalised as a (partial) function from identifiers to values. Hence, enumerating all threads or comparing two stores is not executable by default. Yet, a finite set of identifier-value pairs could easily store such state information, which is normally modified point-wisely. Explicitly using associative lists in one's formalisation, however, incurs a lot of work because one state has in general multiple representations and AC1 unification is not supported.

For such kind of data, we propose to use a new type FinFun of total functions that are constant except for finitely many points. They generalise maps, which formally are total functions of type $'a \Rightarrow {'b}\ option$ that map to *None* ("undefined") almost everywhere, in two ways: First, they can replace (total) functions of arbitrary type $'a \Rightarrow {'b}$. Second, their default value is not fixed to a predetermined value (like *None*). Our main technical contributions are:[1]

1. On the code level, every FinFun is represented as explicit data via two datatype constructors: constant FinFuns and pointwise update (cf. Sec. 2). *quickcheck* is set up for FinFuns and working.
2. Inside the logic, FinFuns feel very much like ordinary functions (e.g. extensionality: $f = g \longleftrightarrow (\forall x.\ f\ x = g\ x)$) and are thus easily integrated into existent formalisations. We demonstrate this in two applications (Sec. 5):
   (a) A formalisation of sets as FinFuns allows sets to be represented explicitly in the generated code.
   (b) We report on our experience in using FinFuns to represent state information for JinjaThreads [12], a semantics for a subset of concurrent Java.
3. Equality tests on, quantification over and other operators on FinFuns are all handled by Isabelle's new code generator (cf. Sec. 3).
4. All equations for code generation have passed through Isabelle's inference kernel, i.e., the trusted code base cannot be compromised by ad-hoc translations where constants in the logic are explicitly substituted by functions of the target language.
5. A recursion combinator allows to directly define functions that are recursive in an argument of FinFun type (Sec. 4).

[1] The FinFun formalisation is available in the Archive of Formal Proofs [13].

FinFuns are a rather restricted class of functions. To represent such functions as associative lists is common knowledge in computer science, but we focus on how to practically hide the problems that such representation issues raise during reasoning without losing the benefits of executability. In Sec. 6, we discuss which functions FinFuns can replace and which not, and compare the techniques and ideas we use with other applications. Isabelle-specific notation is defined in appendix A.

2 Type Definition and Basic Properties

To start with, we construct the new type $'a \Rightarrow_f 'b$ for FinFuns. This type contains all functions from $'a$ to $'b$ which map only finitely many points $a :: 'a$ to some value other than some constant $b :: 'b$, i.e. are constant except for finitely many points. We show that all elements of this type can be built from two constructors: The everywhere constant FinFun and pointwise update of a FinFun (Sec. 2.1). Code generated for operators on FinFuns will be recursive via these two kernel functions (cf. Sec. 2.2).

In Isabelle/HOL, a new type is declared by specifying a non-empty carrier set as a subset of an already existent type. The new type for FinFuns is isomorphic to the set of functions that deviate from a constant at only finitely many points:

$$\textbf{typedef}\ ('a,'b)\ \mathit{finfun} = \{f::'a \Rightarrow 'b \mid \exists\, b.\ \mathit{finite}\ \{a \mid f\ a \neq b\}\}$$

Apart from the new type $('a, 'b)$ *finfun* (written $'a \Rightarrow_f 'b$), this introduces the set $\mathit{finfun} :: ('a \Rightarrow 'b)\ \mathit{set}$ given on the right-hand side and the two bijection functions *Abs-finfun* and *Rep-finfun* between the sets $\mathit{UNIV} :: ('a \Rightarrow_f 'b)\ \mathit{set}$ and *finfun* such that *Rep-finfun* is surjective and they are inverses of each other:

$$\mathit{Rep\text{-}finfun}\ \hat{f} \in \mathit{finfun} \tag{1}$$

$$\mathit{Abs\text{-}finfun}\ (\mathit{Rep\text{-}finfun}\ \hat{f}) = \hat{f} \tag{2}$$

$$f \in \mathit{finfun} \longrightarrow \mathit{Rep\text{-}finfun}\ (\mathit{Abs\text{-}finfun}\ f) = f \tag{3}$$

For clarity, we decorate all variable identifiers of FinFun type $'a \Rightarrow_f 'b$ with a hat ˆ to distinguish them from those of ordinary function type $'a \Rightarrow 'b$. Note that the default value b of the function, to which it does not map only finitely many points, is *not* stored in the type elements themselves. In case $'a$ is infinite, any such b is uniquely determined and would therefore be redundant. If not, $\{a \mid f\ a \neq b\}$ is finite for all $f::'a \Rightarrow 'b$ and $b::'b$, i.e. $\mathit{finfun} = \mathit{UNIV}$. Moreover, if that default value was fixed, then equality on $'a \Rightarrow_f 'b$ would not be as expected, cf. (5).

The function *finfun-default* $\hat{f}$ returns the default value of $\hat{f}$ for infinite domains. For finite domains, we fix it to *undefined* which is an arbitrary (but fixed) constant to represent undefinedness in Isabelle:

$$\mathit{finfun\text{-}default}\ \hat{f} \equiv \textsf{if}\ \mathit{finite}\ \mathit{UNIV}\ \textsf{then}\ \mathit{undefined}\ \textsf{else}\ \iota b.\ \mathit{finite}\ \{a \mid \mathit{Rep\text{-}finfun}\ \hat{f}\ a \neq b\}$$

2.1 Kernel Functions for FinFuns

Having manually defined the type, we now show that every FinFun can be generated from two kernel functions similarly to a **datatype** element from its constructors: The constant function and pointwise update. For $b{::}'b$, let $K^f\, b{::}'a \Rightarrow_f\ 'b$ represent the FinFun that maps everything to b. It is defined by lifting the constant function $\lambda x{::}'a.\ b$ via *Abs-finfun* to the FinFun type. Similarly, pointwise update *finfun-update*, written $\_(\_ :=_f \_)$, is defined in terms of pointwise function update on ordinary functions:

$$K^f\, b \equiv \textit{Abs-finfun}\ (\lambda x.\ b) \quad \text{and} \quad \hat{f}(a :=_f b) \equiv \textit{Abs-finfun}\ ((\textit{Rep-finfun}\ \hat{f})(a := b))$$

Note that these two kernel functions replace λ-abstraction of ordinary functions. Since the code generator will internally use these two constructors to represent FinFuns as data objects, proper λ-abstraction (via *Abs-finfun*) is not executable and is therefore deprecated. Consequently, all executable operators on FinFuns are to be defined (recursively) in terms of these two kernel functions. On the logic level, λ-abstraction is of course available via *Abs-finfun*, but it will be tedious to reason about such functions: Arbitrary λ-abstraction does not guarantee the finiteness constraint in the type definition for $'a \Rightarrow_f\ 'b$, hence this constraint must always be shown separately.

We can now already define what function application on $'a \Rightarrow_f\ 'b$ will be, namely *Rep-finfun*. To facilitate replacing ordinary functions with FinFuns in existent formalisations, we write function applications as a postfix subscript $_f$: $\hat{f}_f\ a \equiv \textit{Rep-finfun}\ \hat{f}\ a$. This directly gives the kernel functions their semantics:

$$(K^f\, b)_f\ a = b \quad \text{and} \quad \hat{f}(a :=_f b)_f\ a' = (\textit{if}\ a = a'\ \textit{then}\ b\ \textit{else}\ \hat{f}_f\ a') \tag{4}$$

Moreover, we already see that extensionality for HOL functions carries over to FinFuns, i.e. $=$ on FinFuns does denote what it intuitively ought to:

$$\hat{f} = \hat{g} \longleftrightarrow (\forall x.\ \hat{f}_f\ x = \hat{g}_f\ x) \tag{5}$$

There are only few characteristic theorems about these two kernel functions. In particular, they are not free constructors, as e.g. the following equalities hold:

$$(K^f\, b)(a :=_f b) = K^f\, b \tag{6}$$

$$\hat{f}(a :=_f b)(a :=_f b') = \hat{f}(a :=_f b') \tag{7}$$

$$a \neq a' \longrightarrow \hat{f}(a :=_f b)(a' :=_f b') = \hat{f}(a' :=_f b')(a :=_f b) \tag{8}$$

This is natural, because FinFuns are meant to behave like ordinary functions and these equalities correspond to the standard ones for pointwise update on ordinary functions. Only $K^f\, \_$ is injective: $(K^f\, b) = (K^f\, b') \longleftrightarrow b = b'$. From a logician's point of view, non-free constructors are not desirable because recursion and case analysis becomes much more complicated. However, the savings in proof automation that extensionality for FinFuns permit are worth the extra effort when it comes to defining operators on FinFuns.

More importantly, these two kernel functions exhaust the type $'a \Rightarrow_f\ 'b$. This is most easily stated by the following induction rule, which is proven by induction on the finite set on which *Rep-finfun* $\hat{g}$ does not take the default value:

$$\frac{\forall\, b.\ P\ (K^f\, b) \qquad \forall\, \hat{f}\ a\ b.\ P\ \hat{f} \longrightarrow P\ \hat{f}(a :=_f\ b)}{P\ \hat{g}} \tag{9}$$

Intuitively, P holds already for all FinFuns $\hat{g}$ if (i) $P\ (K^f\, b)$ holds for all constant FinFuns $K^f\, b$ and (ii) whenever $P\ \hat{f}$ holds, then $P\ \hat{f}(a :=_f\ b)$ holds, too. From this, a case distinction theorem is easily derived:

$$(\exists\, b.\ \hat{g} = (K^f\, b)) \vee (\exists\, \hat{f}\ a\ b.\ \hat{g} = \hat{f}(a :=_f\ b)) \tag{10}$$

Both induction rule and case distinction theorem are weak in the sense that the $\hat{f}$ in the case for point-wise update is quantified without further constraints. Since $K^f\, \_$ and pointwise update are not distinct – cf. (6), proofs that do case analysis on FinFuns must always handle both cases even for constant FinFuns. Stronger induction and case analysis theorems could, however, be derived.

2.2 Representing FinFuns in the Code Generator

As mentioned above, the code generator represents FinFuns as a datatype with constant FinFun and pointwise update as (free) constructors. In Haskell, e.g., the following code is generated:

```
data Finfun a b = Finfun_update_code (Finfun a b) a b
                | Finfun_const b;
```

For efficiency reasons, we do not use *finfun-update* as a constructor for the `Finfun` datatype, as overwritten updates then would not get removed, the function's representation would keep growing. Instead, the HOL constant *finfun-update-code*, denoted $\_(\!|\_ :=_f\ \_|\!)$, is employed, which is semantically equivalent: $\hat{f}(\!|a :=_f\ b|\!) \equiv \hat{f}(a :=_f\ b)$. The code for *finfun-update*, however, is generated from (11) and (12):

$$(K^f\, b)(a :=_f\ b') = \mathit{if}\ b = b'\ \mathit{then}\ K^f\, b\ \mathit{else}\ (K^f\, b)(\!|a :=_f\ b'|\!) \tag{11}$$

$$\hat{f}(\!|a :=_f\ b|\!)(a' :=_f\ b') = \mathit{if}\ a = a'\ \mathit{then}\ \hat{f}(a :=_f\ b')\ \mathit{else}\ \hat{f}(a' :=_f\ b')(\!|a :=_f\ b|\!) \tag{12}$$

```
finfun_update :: forall a b. (Eq a, Eq b) =>
                 Finfun a b -> a -> b -> Finfun a b;
finfun_update (Finfun_update_code f a b) a' b' =
  (if eqop a a' then finfun_update f a b'
    else Finfun_update_code (finfun_update f a' b') a b);
finfun_update (Finfun_const b) a b' =
  (if eqop b b' then Finfun_const b
    else Finfun_update_code (Finfun_const b) a b');
```

where `eqop` is the HOL equality operator given by `eqop a = (\ b -> a == b);`. Hence, an update with $\_(\_ :=_f\ \_)$ is checked against all other updates, all overwritten updates are thereby removed, and inserted only if it does not update to the default value. Using $\_(\_ :=_f \_)$ in the logic ensures that on the code level, every FinFun is stored with as few updates as possible given the fixed default value.[2]

Let, e.g., $\hat{f} = (K^f 0)(\!|1 :=_f 5|\!)(\!|2 :=_f 6|\!)$. When $\hat{f}$ is updated at 1 to 0, $\hat{f}(1 :=_f 0)$ evaluates on the code level to $(K^f 0)(\!|2 :=_f 6|\!)$, where all redundant updates at 1 have been removed. If the explicit code update function had been used instead, the last update would have been added to the list of updates: $\hat{f}(\!|1 :=_f 0|\!)$ evaluates to $(K^f 0)(\!|1 :=_f 5|\!)(\!|2 :=_f 6|\!)(\!|1 :=_f 0|\!)$. Exactly this problem of superfluous updates would occur if $\_(\_ :=_f \_)$ was directly used as a constructor in the exported code.

In case this optimisation is undesired, one can use *finfun-update-code* instead of *finfun-update*. Redundant updates in the representation on the code level can subsequently be deleted by invoking the *finfun-clearjunk* operator: Semantically, this is the identity function: *finfun-clearjunk* $\equiv$ *id*, but it is implemented using the following to equations that remove all redundant updates:

$$\textit{finfun-clearjunk}\ (K^f b) = (K^f b) \quad \text{and} \quad \textit{finfun-clearjunk}\ \hat{f}(\!|a :=_f b|\!) = \hat{f}(a :=_f b)$$

Consequently, every function that is defined recursively on FinFuns must provide two such equations for $K^f\,\_$ and $\_(\!|\_ :=_f \_|\!)$ for being executable. For function application, e.g., those from (4) are used with *finfun-update* being replaced by *finfun-update-code*.

For *quickcheck*, we have installed a sampling function that randomly creates a FinFun which has been updated at a few random points to random values. Hence, *quickcheck* can now both evaluate operators involving FinFuns and sample random values for the free variables of FinFun type in a conjecture.

3 Operators for FinFuns

In the previous section, we have shown how FinFuns are defined in Isabelle/HOL and how they are implemented in code. This section introduces more executable operators on FinFuns moving from basic ones towards executable equality.

3.1 Function Composition

The most important operation on functions and FinFuns alike – apart from application – is composition. It creates new FinFuns from old ones without losing executability: Every ordinary function $g{::}'b \Rightarrow {}'c$ can be composed with a FinFun $\hat{f}$ of type $'a \Rightarrow_f {}'b$ to produce another FinFun $g \circ_f \hat{f}$ of type $'a \Rightarrow_f {}'c$. The operator $\circ_f$ is defined like the kernel functions via *Abs-finfun* and *Rep-finfun*:

$$g \circ_f \hat{f} \equiv \textit{Abs-finfun}\ (g \circ \textit{Rep-finfun}\ \hat{f})$$

To the code generator, two recursive equations are provided:

$$g \circ_f (K^f c) = (K^f g\ c) \quad \text{and} \quad g \circ_f \hat{f}(\!|a :=_f b|\!) = (g \circ_f \hat{f})(\!|a :=_f g\ b|\!) \tag{13}$$

[2] Minimal is relative to the default value in the representation (which need not coincide with *finfun-default*) – i.e. this does not include the case where changing this default value would require less updates. $(K^f 0)(\textit{True} :=_f 1)(\textit{False} :=_f 1)$ of type $\textit{bool} \Rightarrow_f \textit{nat}$, e.g., is stored as $(K^f 0)(\!|\textit{False} :=_f 1|\!)(\!|\textit{True} :=_f 1|\!)$, whereas $K^f 1$ would also do.

$\circ_f$ is more versatile than composition on FinFuns only, because ordinary functions can be written directly thanks to λ abstraction. Yet, a FinFun $\hat{g}$ is equally easily composed with another FinFun $\hat{f}$ if we convert the first one back to ordinary functions: $\hat{g}_f \circ_f \hat{f}$. However, composing a FinFun with an ordinary function is not as simple. Although the definition is again straightforward:

$$\hat{f} \;{}_f{\circ}\; g \equiv \textit{Abs-finfun}\ (\textit{Rep-finfun}\ \hat{f} \circ g),$$

reasoning about ${}_f{\circ}$ is more difficult: Take, e.g., $\hat{f} = (K^f 2)(1 :=_f 1)$ and $g = (\lambda x.\ x\ \textit{mod}\ 2)$. Then, $\hat{f} \;{}_f{\circ}\; g$ ought to be the function that maps even numbers to 2 and odd ones to 1, which is not a FinFun any more. Hence, (3) can no longer be used to reason about $\hat{f} \;{}_f{\circ}\; g$, so nothing nontrivial can be deduced about $\hat{f} \;{}_f{\circ}\; g$.

If g is injective (written $\textit{inj}\ g$), then $\hat{f} \;{}_f{\circ}\; g$ behaves as expected on updates:

$$\hat{f}(b :=_f c) \;{}_f{\circ}\; g = (\textsf{\textit{if}}\ b \in \textit{range}\ g\ \textsf{\textit{then}}\ (\hat{f} \;{}_f{\circ}\; g)(g^{-1}\ b :=_f c)\ \textsf{\textit{else}}\ \hat{f} \;{}_f{\circ}\; g), \tag{14}$$

where $\textit{range}\ g$ denotes the range of g and g^{-1} is the inverse of g. Clearly, both $b \in \textit{range}\ g$ and $g^{-1}\ b$ are not executable for arbitrary g, so this conditional equality is not suited for code generation. If terms involving ${}_f{\circ}$ are to be executed, the above equation must be specialised to a specific g to become executable. The constant case is trivial for all g and need not be specialised: $(K^f c) \;{}_f{\circ}\; g = (K^f c)$.

This composition operator is good for reindexing the domain of a FinFun: Suppose, e.g., we need $\hat{h}_f\ x = \hat{f}_f\ (x + a)$ for some $a{::}\textit{int}$, then $\hat{h}$ could be defined as $\hat{h} \equiv \hat{f} \;{}_f{\circ}\; g$ with $g = (\lambda x.\ x + a)$. Clearly, $\textit{inj}\ g$, $\textit{range}\ g = \textit{UNIV}$ and $g^{-1} = (\lambda x.\ x - a)$, so (14) simplifies to $\hat{f}(b :=_f c) \;{}_f{\circ}\; g = (\hat{f} \;{}_f{\circ}\; g)(b - a :=_f c)$. Unfortunately, the code generator cannot deal with such specialised recursion equations where the second parameter of $\_ \;{}_f{\circ}\; \_$ is instantiated to g, so a new constant $\textit{shift}\ \hat{f}\ a \equiv \hat{f} \;{}_f{\circ}\; (\lambda x.\ x + a)$ must be introduced for the code generator with the recursion equations $\textit{shift}\ (K^f b)\ a = (K^f b)$ and $\textit{shift}\ \hat{f}(\!|a' :=_f b|\!)\ a = (\textit{shift}\ \hat{f}\ a)(a' - a :=_f b)$.

3.2 FinFuns and Pairs

Apart from composing FinFuns one after another, one often has to "run" FinFuns in parallel, i.e. evaluate both on the same argument and return both results as a pair. For two functions f and g, this is done by the term $\lambda x.\ (f\ x,\ g\ x)$. For two FinFuns $\hat{f}$ and $\hat{g}$, λ abstraction is not executable, but an appropriate operator $(\hat{f},\ \hat{g})^f$ is easily defined as

$$(\hat{f},\ \hat{g})^f \equiv \textit{Abs-finfun}\ (\lambda x.\ (\textit{Rep-finfun}\ \hat{f}\ x,\ \textit{Rep-finfun}\ \hat{g}\ x)).$$

This operator is most useful when two FinFuns are to be combined pointwise by some combinator h, which is then $\circ_f$-composed with this diagonal operator: Suppose, e.g., that $\hat{f}$ and $\hat{g}$ are two integer FinFuns and we need their pointwise sum, which is $(\lambda(x,\ y).\ x + y) \circ_f (\hat{f},\ \hat{g})^f$, i.e. h is uncurried addition. The code equations are straight forward again:

$$(K^f b,\ K^f c)^f = K^f (b,\ c) \tag{15}$$

$$(K^f b,\ \hat{g}(\!|a :=_f c|\!))^f = (K^f b,\ \hat{g})^f (\!|a :=_f (b,\ c)|\!) \tag{16}$$

$$(\hat{f}(\!|a :=_f b|\!),\ \hat{g})^f = (\hat{f},\ \hat{g})^f (a :=_f (b,\ \hat{g}_f\ a)) \tag{17}$$

3.3 Executable Quantifiers

Quantifiers in Isabelle/HOL are defined as higher-order functions. The universal quantifier *All* is defined by $\mathit{All}\ P \equiv P = (\lambda x.\ \mathit{True})$ where P is a predicate and the binder notation $\forall x.\ P\ x$ is then just syntactic sugar for $\mathit{All}\ (\lambda x.\ P\ x)$. This also explains the error message of the code generator from Sec. 1. However, without λ-abstraction, there is no such nice notation for FinFuns, but the operator *finfun-All* for universal quantification over FinFun predicates is straightforward: $\mathit{finfun\text{-}All}\ \hat{P} \equiv \forall x.\ \hat{P}_f\ x$.

Clearly, reducing universal quantification over FinFuns to *All* does not help with code generation, which was the main point in introducing FinFuns in the first place. However, we can exploit the explicit representation of $\hat{P}$. To that end, a more general operator *ff-All* of type $'a\ \mathit{list} \Rightarrow\ 'a \Rightarrow_f \mathit{bool} \Rightarrow \mathit{bool}$ is necessary which ignores all points of $\hat{P}$ that are listed in the first argument:

$$\mathit{ff\text{-}All}\ \mathit{as}\ \hat{P} \equiv \forall a.\ a \in \mathit{set}\ \mathit{as} \vee \hat{P}_f\ a$$

Clearly, $\mathit{finfun\text{-}All} = \mathit{ff\text{-}All}\ []$ holds. The extra list *as* keeps track of which points have already been updated and can be ignored in recursive calls:

$$\mathit{ff\text{-}All}\ \mathit{as}\ (K^f\, b) \longleftrightarrow b \vee \mathit{set}\ \mathit{as} = \mathit{UNIV} \tag{18}$$

$$\mathit{ff\text{-}All}\ \mathit{as}\ \hat{P}(\!|a :=_f b|\!) \longleftrightarrow (a \in \mathit{set}\ \mathit{as} \vee b) \wedge \mathit{ff\text{-}All}\ (a{\cdot}\mathit{as})\ \hat{P} \tag{19}$$

In the recursive case, the update a to b must either be overwritten by a previous update ($a \in \mathit{set}\ \mathit{as}$) or have b equal to *True*. Then, for the recursive call, a is added to the list *as* of visited points. In the constant case, either the constant is *True* itself or all points of the domain $'a$ have been updated ($\mathit{set}\ \mathit{as} = \mathit{UNIV}$).

Via $\mathit{finfun\text{-}All} = \mathit{ff\text{-}All}\ []$, *finfun-All* is now executable, provided the test $\mathit{set}\ \mathit{as} = \mathit{UNIV}$ can be operationalised. Since $\mathit{as}{::}'a\ \mathit{list}$ is a (finite) list, $\mathit{set}\ \mathit{as}$ is by construction always finite. Thus, for infinite domains $'a$, this test always fails. Otherwise, if $'a$ is finite, such a test can be easily implemented.

Note that this distinction can be directly made on the basis of type information. Hence, we shift this subtle distinction into a type class such that the code automatically picks the right implementation for $\mathit{set}\ \mathit{as} = \mathit{UNIV}$ based on type information. Axiomatic type classes [7] allow for HOL constants being safely overloaded for different types and are correctly handled by Haftmann's code generator [6]. If the output language supports type classes like e.g. Haskell does, this feature is directly employed. Otherwise, functions in generated code are provided with an additional dictionary parameter that selects the appropriate implementation for overloaded constants at runtime.

For our purpose, we introduce a new type class *card-UNIV* with one parameter *card-UNIV* and the axiom that $\mathit{card\text{-}UNIV} :: 'a\ \mathit{itself} \Rightarrow \mathit{nat}$ returns the cardinality of $'a$'s universe:

$$\mathit{card\text{-}UNIV}\ x = \mathit{card}\ \mathit{UNIV} \tag{20}$$

By default, the cardinality of a type's universe is just a natural number of type *nat*, which itself is not related to $'a$ at all. Hence, *card-UNIV* takes an artificial parameter of type $'a\ \mathit{itself}$, where *itself* represents types at the level of values: $\mathit{TYPE}('a)$ is the value associated with the type $'a$.

As every HOL type is inhabited, *card-UNIV TYPE('a)* can indeed be used to discriminate between types with finite and infinite universes by testing against 0:

$$\textit{finite}\ (\textit{UNIV}::'a\ \textit{set}) \longleftrightarrow 0 < \textit{card-UNIV TYPE}('a)$$

Moreover, the test *set as* = *UNIV* can now be written as *is-list-UNIV as* with

$\textit{is-list-UNIV as} \equiv$
$\textsf{let}\ c = \textit{card-UNIV TYPE}('a)\ \textsf{in}\ \textsf{if}\ c = 0\ \textsf{then}\ \textit{False}\ \textsf{else}\ |\textit{remdups as}| = c$

where *remdups as* removes all duplicates from the list *as*.

Note that the constraint (20) on the type class parameter *card-UNIV*, which is to be overloaded, is purely definitional. Thus, every type could be made member of the type class *card-UNIV* by instantiating *card-UNIV* to $\lambda a.$ *card UNIV*. However, for executability, it must be instantiated such that the code generator can generate code for it. This has been done for the standard HOL types like *unit*, *bool*, *char*, *nat*, *int*, and *'a list*, for which it is straightforward if one remembers that *card* $A = 0$ for all infinite sets A. For the type *bool*, e.g., *card-UNIV* a $\equiv 2$ for all a::*bool itself*. The cardinality of the universe for polymorphic type constructors like e.g. $'a \times 'b$ is computed by recursion on the type parameters:

$$\textit{card-UNIV TYPE}('a \times 'b) = \textit{card-UNIV TYPE}('a) \cdot \textit{card-UNIV TYPE}('b)$$

We have similarly instantiated *card-UNIV* for the type constructors $'a \Rightarrow 'b$, *'a option* and $'a + 'b$.

As we have the universal quantifier *finfun-All*, the executable existential quantifier is straightforward by duality: *finfun-Ex* $\hat{P} \equiv \neg$ *finfun-All* (*Not* $\circ_f$ $\hat{P}$). As before, the pretty-print syntax $\exists x.\ P\ x$ for *Ex* $(\lambda x.\ P\ x)$ in HOL cannot be transferred to FinFuns because λ-abstraction is not suited for code generation.

3.4 Executable Equality on FinFuns

Our second main goal with FinFuns, besides executable quantifiers, is executable equality tests on FinFuns. Extensionality – cf. (5) – reduces function equality to equality on every argument. However, (5) does not directly yield an implementation because it uses the universal quantifier *All* for ordinary HOL predicates, but some rewriting does the trick:

$$\hat{f} = \hat{g} \longleftrightarrow \textit{finfun-All}\ ((\lambda(x, y).\ x = y) \circ_f (\hat{f}, \hat{g})^f) \tag{21}$$

By instantiating the HOL type class *eq* appropriately, the equality operator = becomes executable and in the generated code, an appropriate equality relation on the datatype is generated. In Haskell, e.g., the equality operator `==` on the type `Finfun a b` then really denotes equality like on the logic level:

```
eq_finfun :: forall a b. (FinFun.Card_UNIV a, Eq a, Eq b) =>
  FinFun.Finfun a b -> FinFun.Finfun a b -> Bool;
eq_finfun f g = FinFun.finfun_All
    (FinFun.finfun_comp (\ (a @ (aa, b)) -> aa == b)
                         (FinFun.finfun_Diag f g));
instance (FinFun.Card_UNIV a, Eq a, Eq b) =>
  Eq (FinFun.Finfun a b) where { (==) = FinFun.eq_finfun; };
```

3.5 Complexity

In this section, we briefly discuss the complexity of the above operators. We assume that equality tests require constant time. For a FinFun $\hat{f}$, let $\#\hat{f}$ denote the number of updates in its code representation. For an ordinary function g, let $\#g$ denote the complexity of evaluating $g\ a$ for any a.

$K^f\ \_$ has constant complexity as it is a `finfun` constructor. Since $\_(\!|\_ :=_f \_|\!)$ automatically removes redundant updates (11, 12), $\hat{f}(\!|\_ :=_f \_|\!)$ is linear in $\#\hat{f}$, and so is application $\hat{f}_f\ \_$ (4). For $g \circ_f \hat{f}$, eq. (13) is recursive in $\hat{f}$ and each recursion step involves $\_(\!|\_ :=_f \_|\!)$ and evaluating g, so the complexity is $\mathcal{O}((\#\hat{f})^2 + \#\hat{f} \cdot \#g)$.

For the product $(\hat{f}, \hat{g})^f$, we get: The base case $(K^f b, \hat{g})^f$ (15, 16) is linear in $\#\hat{g}$ and we have $\#(K^f b, \hat{g})^f = \#\hat{g}$. An update in the first parameter $(\hat{f}(\!|a :=_f b|\!), \hat{g})^f$ (17) executes $\hat{g}_f\ a$ ($\mathcal{O}(\#\hat{g})$), the recursive call and the update ($\mathcal{O}(\#(\hat{f}, \hat{g})^f)$). Since there are $\#\hat{f}$ recursive calls and $\#(\hat{f}, \hat{g})^f \leq \#\hat{f} + \#\hat{g}$, the total complexity is bound by $\mathcal{O}(\#\hat{f} \cdot (\#\hat{f} + \#\hat{g}))$.

Since *finfun-All* is directly implemented in terms of *ff-All*, it is sufficient to analyse the latter's complexity: The base case (18) essentially executes *is-list-UNIV*. If we assume that the cardinality of the type universe is computed in constant time, *is-list-UNIV as* is bound by $\mathcal{O}(|as|^2)$ since *remdups as* takes $\mathcal{O}(|as|^2)$ steps. In case of an update (19), the updated point is checked against the list *as* ($\mathcal{O}(|as|)$) and the recursive call is executed with the list *as* being one element longer, i.e. $|as|$ grows by one for each recursive call. As there are $\#\hat{P}$ many recursive calls, *ff-All as* $\hat{P}$ has complexity $\#\hat{P} \cdot \mathcal{O}(\#\hat{P} + |as|) + \mathcal{O}((\#\hat{P} + |as|)^2) = \mathcal{O}((\#\hat{P} + |as|)^2)$. Hence, *finfun-All* $\hat{P}$ has complexity $\mathcal{O}((\#\hat{P})^2)$.

Equality on FinFuns $\hat{f}$ and $\hat{g}$ is then straightforward (21): $(\hat{f}, \hat{g})^f$ is in $\mathcal{O}(\#\hat{f} \cdot (\#\hat{f} + \#\hat{g}))$. Composing this with $\lambda(x, y).\ x = y$ takes $\mathcal{O}((\#(\hat{f}, \hat{g})^f)^2) \subseteq \mathcal{O}((\#\hat{f} + \#\hat{g})^2)$. Finally, executing *finfun-All* is quadratic in $\#((\lambda(x, y).\ x = y) \circ_f (\hat{f}, \hat{g})^f) \leq \#(\hat{f}, \hat{g})^f$. In total, $\hat{f} = \hat{g}$ has complexity $\mathcal{O}((\#\hat{f} + \#\hat{g})^2)$.

4 A Recursion Combinator

In the previous section, we have presented several operators on FinFuns that suffice for most purposes, cf. Sec. 5. However, we had to define function composition with FinFuns on either side and operations on products manually by going back to the type's carrier set *finfun* via *Rep-finfun* and *Abs-finfun*. This is not only inconvenient, but also loses the abstraction from the details of the finite set of updated points that FinFuns provide. In particular, one has to derive extra recursion equations for the code generator and prove each of them correct.

Yet, the induction rule (9) states that the recursive equations uniquely determine any function that satisfies these. Operations on FinFuns could therefore be defined by primitive recursion similarly to datatypes (cf. [2]). Alas, the two FinFun constructors are not free, so not every pair of recursive equations does indeed define a function. It might also well be the case that the equations are contradictory: For example, suppose we want to define a function *count* that counts the number of updates, i.e. *count* $(K^f c) = 0$ and *count* $\hat{f}(\!|a :=_f b|\!) =$ *count* $\hat{f} + 1$. Such a function does not exist for FinFuns in Isabelle, although it could be defined in Haskell to, e.g., compute extra-logic data such as memory consumption. Take, e.g., $\hat{f} \equiv (K^f 0)(\!|0 :=_f 0|\!)$. Then, *count* $\hat{f} =$ *count* $(K^f 0) + 1 = 1$, but $\hat{f} = (K^f 0)$ by (6) and thus *count* $\hat{f} = 0$ would equally have to hold, because equality is congruent w.r.t. function application, a contradiction.

4.1 Lifting Recursion from Finite Sets to FinFuns

More abstractly, the right hand side of the recursive equations can be considered as a function: For the constant case, such a function $c{::}'b \Rightarrow {}'c$ takes the constant value of the FinFun and evaluates to the right hand side. In the recursive case, $u{::}'a \Rightarrow {}'b \Rightarrow {}'c \Rightarrow {}'c$ takes the point of the update, the new value at that point and the result of the recursive call. In this section, we define a combinator *finfun-rec* that takes c and u and defines the corresponding operator on FinFuns, similarly to the primitive recursion combinators that are automatically generated for datatypes. That is, *finfun-rec* must satisfy (22) and (23), subject to certain well-formedness conditions on c and u, which will be examined in Sec. 4.2.

$$\textit{finfun-rec}\ c\ u\ (K^f b) = c\ b \tag{22}$$

$$\textit{finfun-rec}\ c\ u\ \hat{f}(\!|a :=_f b|\!) = u\ a\ b\ (\textit{finfun-rec}\ c\ u\ \hat{f}) \tag{23}$$

The standard means in Isabelle for defining recursive functions, namely **recdef** and the function package [10], are not suited for this task because both need a termination proof, i.e. a well-founded relation in which all recursive calls always decrease. Since $K^f\,\_$ and $\_(\!|\_ :=_f \_|\!)$ are not free constructors, there is no such termination order for (22) and (23). Hence, we define *finfun-rec* by recursion on the finite set of updated points using the recursion operator *fold* for finite sets:

$\textit{finfun-rec}\ c\ u\ \hat{f} \equiv$
$\textsf{let}\ b = \textit{finfun-default}\ \hat{f};$
$\quad g = (\iota g.\ \hat{f} = \textit{Abs-finfun}\ (\textit{map-default}\ b\ g) \wedge \textit{finite}\ (\textit{dom}\ g) \wedge b \notin \textit{ran}\ g)$
$\textsf{in}\ \textit{fold}\ (\lambda a.\ u\ a\ (\textit{map-default}\ b\ g\ a))\ (c\ b)\ (\textit{dom}\ g)$

In the *let* expression, $\hat{f}$ is unpacked into its default value b (cf. Sec. 2) and a partial function $g{::}'a \rightharpoonup {}'b$ such that $\hat{f} = \textit{Abs-finfun}\ (\textit{map-default}\ b\ g)$ and the finite domain of g contains only points at which $\hat{f}$ differs from its default value b, i.e. g stores precisely the updates of $\hat{f}$. Then, the update function u is folded over the finite set of points *dom* g where $\hat{f}$ does not take its default value b.

All FinFun operators that we have defined in Sec. 3 via *Abs-finfun* and *Rep-finfun* can also be defined directly via *finfun-rec*. For example, the functions for $\circ_f$ directly show up in the recursive equations from (13):

$$g \circ_f \hat{f} \equiv \textit{finfun-rec}\ (\lambda b.\ K^f g\ b)\ (\lambda a\ b\ \hat{f}.\ \hat{f}(\!|a :=_f g\ b|\!))\ \hat{f}.$$

4.2 Well-formedness Conditions

Since all functions in HOL are total, *finfun-rec* $c\ u$ is defined for every combination of c and u. Any nontrivial property of *finfun-rec* is only provable if u is left-commutative because *fold* is unspecified for other functions. Thus, the next step is to establish conditions on the FinFun level that ensure (22) and (23). It turns out that four are sufficient:

$$u\ a\ b\ (c\ b) = c\ b \tag{24}$$

$$u\ a\ b''\ (u\ a\ b'\ (c\ b)) = u\ a\ b''\ (c\ b) \tag{25}$$

$$a \neq a' \longrightarrow u\ a\ b\ (u\ a'\ b'\ d) = u\ a'\ b'\ (u\ a\ b\ d) \tag{26}$$

$$\textit{finite UNIV} \longrightarrow \textit{fold}\ (\lambda a.\ u\ a\ b')\ (c\ b)\ \textit{UNIV} = c\ b' \tag{27}$$

Eq. (24), (25), and (26) naturally reflect the equalities between the constructors from (6), (7), and (8), respectively. It is sufficient to restrict overwriting updates (25) to constant FinFuns because the general case directly follows from this by induction and (26). The last equation (27) arises from the identity

$$\textit{finite UNIV} \longrightarrow \textit{fold}\ (\lambda a\ \hat{f}.\ \hat{f}(a :=_f\ b'))\ (K^f b)\ \textit{UNIV} = (K^f b'). \tag{28}$$

Eq. (24), (25), and (26) are sufficient for proving (23). For a FinFun operator like $\circ_f$, these constraints must be shown for specific c and u, which is usually completely automatic. Even though (27), which is required to deduce (22), must usually be proven by induction, this normally is also automatic, because for finite types $'a$, $'a \Rightarrow\ 'b$ and $'a \Rightarrow_f\ 'b$ are isomorphic via *Abs-finfun* and *Rep-finfun*.

5 Applications

In this section, we present two applications for FinFuns to demonstrate that the operations from Sec. 3 form a reasonably complete set of abstract operations.

1. They can be used to represent sets as predicates with the standard operations all being executable: membership and subset test, union, intersection, complement and bounded quantification.
2. FinFuns have been inspired by the needs of JinjaThreads [12], which is a formal semantics of multithreaded Java in Isabelle. We show how FinFuns prove essential on the way to generating an interpreter for concurrent Java.

5.1 Representing Sets with Finfuns

In Isabelle 2008, the proper type $'a$ *set* for sets has been removed in favour of predicates of type $'a \Rightarrow$ *bool* to eliminate redundancies in the implementation and in the library. As a consequence, Isabelle's new code generator is no longer able to generate code for sets as before: A finite set had been coded as the list of its elements. Hence, e.g. the complement operator has not been executable because the complement of a finite set might no longer be a finite set. Neither are collections of the form $\{a \mid P\ a\}$ suited for code generation.

Since FinFuns are designed for code generation, they can be used for representing sets in explicit form without explicitly introducing a set type of its own. FinFun set operations like membership and inclusion test, union, intersection and even complement are straightforward using $\circ_f$. As before, these operators are decorated with f subscripts to distinguish them from their analogues on sets:

$$\hat{f} \subseteq_f \hat{g} \equiv \textit{finfun-All}\ ((\lambda(x, y).\ x \longrightarrow y) \circ_f (\hat{f}, \hat{g})^f) \qquad -\hat{f} \equiv (\lambda b.\ \neg\ b) \circ_f \hat{f}$$

$$\hat{f} \cup_f \hat{g} \equiv (\lambda(x, y).\ x \vee y) \circ_f (\hat{f}, \hat{g})^f \qquad \hat{f} \cap_f \hat{g} \equiv (\lambda(x, y).\ x \wedge y) \circ_f (\hat{f}, \hat{g})^f$$

Obviously, these equations can be directly translated into executable code.

However, if we were to reason with them directly, most theorems about sets (as predicates) would have to be replicated for FinFuns. Although this would be straightforward, loads of redundancy would be reintroduced this way. Instead, we propose to inject FinFun sets via $_f$ into ordinary sets and use the standard operations on sets to work with them. The code generator is set up such that it preprocesses all equations for code generation and automatically replaces set operations with their FinFun equivalents by unfolding equations such as $A_f \subseteq B_f \longleftrightarrow A \subseteq_f B$ and $A_f \cup B_f = (A \cup_f B)_f$. This approach works for *quickcheck*, too.

Besides the above operations, bounded quantification is also straightforward:

$$\textit{finfun-Ball}\ \hat{A}\ P \equiv \forall x \in \hat{A}_f.\ P\ x \quad \text{and} \quad \textit{finfun-Bex}\ \hat{A}\ P \equiv \exists x \in \hat{A}_f.\ P\ x$$

Clearly, they are not executable right away. Take, e.g., $\hat{A} = (K^f\, \textit{True})$, i.e. the universal set, then $\textit{finfun-Ball}\ \hat{A}\ P \longleftrightarrow (\forall x.\ P\ x)$, which is undecidable if x ranges over an infinite domain. However, if we go for partial correctness, correct code can be generated: Like for the universal quantifier *finfun-All* for FinFun predicates (cf. Sec. 3.3), *ff-Ball* is introduced which takes an additional parameter *xs* to remember the list of points which have already been checked at previous calls.

$$\textit{ff-Ball}\ \textit{xs}\ \hat{A}\ P \equiv \forall a \in \hat{A}_f.\ a \in \textit{set}\ \textit{xs} \vee P\ a.$$

This now permits to set up recursive equations for the code generator:

$$\textit{ff-Ball}\ \textit{xs}\ (K^f\, b)\ P \longleftrightarrow \neg\, b \vee \textit{set}\ \textit{xs} = \textit{UNIV} \vee \textit{loop}\ (\lambda u.\ \textit{ff-Ball}\ \textit{xs}\ (K^f\, b)\ P)$$
$$\textit{ff-Ball}\ \textit{xs}\ \hat{A}(\!|a :=_f b|\!)\ P \longleftrightarrow (a \in \textit{set}\ \textit{xs} \vee (b \longrightarrow P\ a)) \wedge \textit{ff-Ball}\ (a \cdot \textit{xs})\ \hat{A}\ P$$

In the constant case, if b is false, i.e. the set is empty, *ff-Ball* holds; similarly, if all elements of the universe have been checked already, this test is again implemented by the overloaded term *is-list-UNIV xs* (Sec. 3.3). Otherwise, one would have to check whether P holds at all points except *xs*, which is not computable for arbitrary P and $'a$. Thus, instead of evaluating its argument, the code for *loop* never terminates. In Isabelle, however, *loop* is simply the *unit*-lifted identity function: $\textit{loop}\ f \equiv f\ ()$. Of course, an exception could equally be raised in place of non-termination. The bounded existential quantifier is implemented analogously.

5.2 JinjaThreads

Jinja [9] is an executable formal semantics for a large subset of Java source-code and bytecode in Isabelle/HOL. JinjaThreads [11] extends Jinja with Java's thread features on both levels. It contains a framework semantics which interleaves the individual threads whose small-step semantics is given to it as a parameter. This framework semantics takes care of all management issues related to threads: The thread pool itself, the lock state, monitor wait sets, spawning and joining a thread, etc. Individual threads communicate via the shared memory with each other and via thread actions like *Lock*, *Unlock*, *Join*, etc. with the framework semantics. At every step, the thread specifies which locks to acquire or release how many times, which thread to create or join on. In our previous work [12], this communication was modelled as a list of such actions, and a lot of pointless work went into identifying permutations of such lists which are semantically equivalent. Therefore, this has been changed such that every lock of type $'l$ now has its own list. Since only finitely many locks need to be changed in any single step, these lists are stored in a FinFun such that checking whether a step's actions are feasible in a given state is executable.

Moreover, in developing JinjaThreads, we have found that most lemmas about the framework semantics contain non-executable assumptions about the thread pool or the lock state, in particular universal quantifiers or predicates defined in terms of them. Therefore, we replaced ordinary functions that model the lock state (type $'l \Rightarrow 't\ lock$) and the thread pool (type $'t \rightharpoonup ('x,\ 'l)\ thread$) with FinFuns. Rewriting the existing proofs took very little effort because mostly, only fs in subscript or superscript had to be added to the proof texts because Isabelle's simplifier and classical reasoner are set up such that FinFuns indeed behave like ordinary functions.

Not to break the proofs, we did not remove the universal quantifiers in the definitions of predicates themselves, but provided simple lemmas to the code generator. For example, *locks-ok ls t las* checks whether all lock requests *las* of thread t can be met in the lock state *ls* and is defined as $\textit{locks-ok}\ ls\ t\ las \equiv \forall l.\ \textit{lock-ok}\ (ls_f\ l)\ t\ (las_f\ l)$, whereas the equation for code generation is

$$\textit{locks-ok}\ ls\ t\ las = \textit{finfun-All}\ ((\lambda(l,\ la).\ \textit{lock-ok}\ l\ t\ la) \circ_f (ls,\ las)^f).$$

Unfortunately, JinjaThreads is not yet fully executable because the semantics of a single thread relies on inductive predicates. Once the code generator will handle these, we will have a certified Jinja virtual machine with concurrency to execute multithreaded Jinja programs as has been done for sequential ones [9].

6 Related Work and Conclusion

Related work. To represent (partial) functions explicitly by a list of point-value pairs is common knowledge in computer science, partial functions $'a \rightharpoonup 'b$ with finite domain have even been formalised as associative lists in the Isabelle/HOL library. However, it is cumbersome to reason with them because one single function has multiple representations, i.e. associative lists are not extensional. Coq and HOL4, e.g., also come with a formalisation of finite maps of their own and both of them fix their default value to *None*. Collins and Syme [4] have already provided a theory of partial functions with finite domain in terms of the everywhere undefined function and pointwise update. Similar to (4), (7), and (8), they axiomatize a type $('a,'b)\ \textit{fmap}$ in terms of abstract operations *Empty*, *Update*, $\textit{Apply} :: ('a,'b)\ \textit{fmap} \Rightarrow 'a \Rightarrow 'b$, and *Domain* and present two models: Maps $'a \rightharpoonup 'b$ with finite domain and associative lists where the order of their elements is determined with Hilbert's choice operator, but neither of these supports code generation. Moreover, equality is not extensional like ours (5), but guarded by the domains. Since these partial functions have an unspecified default value that is implicitly fixed by the codomain type and the model, they cannot be used for almost everywhere constant functions where the default value may differ from function to function. Consequently, (28) is not expressible in their setting.

Recursion over non-free kernel functions is also a well-known concept: Nipkow and Paulson [14], e.g., define a *fold* operator for finite sets which are built from the empty set and insertion of one element. However, they do not introduce a new type for finite sets, so all equations are guarded by the predicate *finite*, i.e. they cannot be leveraged by the code generator.

Nominal Isabelle [16] is used to facilitate reasoning about α-equivalent terms with binders, where the binders are non-free term constructors. The HOL type for terms is obtained by quotienting the datatype with the (free) term constructors w.r.t. α-equivalence classes. Primitive-recursive definitions must then be shown compatible with α-equivalence using a notion of freshness [17]. It is tempting to define the FinFun type universe similarly as the quotient of the datatype with constructors $K^f\,\_$ and $\_(\!|\_ :=_f \_|\!)$ w.r.t. the identities (6), (7), (8), and (28), because this would settle exhaustion, induction and recursion almost automatically. However, this construction is not directly possible because (28) cannot be expressed as an equality of kernel functions. Instead, we have defined the carrier set *finfun* directly in terms of the function space and established easy, sufficient (and almost necessary) conditions for recursive definitions being well-formed.

Conclusion. FinFuns generalise finite maps by continuing them with a default value in the logic, but for the code generator, they are implemented like associative lists which suffer from multiple representations for a single function. Thus, they bridge the gap between easy reasoning and these implementation issues arising from functions as data: They are as easy to use as ordinary functions. By not fixing a default value (like *None* for maps), we have been able to easily apply them to very diverse settings.

We have decided to restrict the FinFun carrier set *finfun* to functions that are constant almost everywhere. Although everything from Sec. 3 would equally work if that restriction was lifted, the induction rule (9) and recursion operator (Sec. 4) would then no longer be available, i.e. the datatype generated by the code generator would not exhaust the type in the logic. Thus, the user could not be sure that every FinFun from his formalisation can be represented as data in the generated code. Conversely, not every operator can be lifted to FinFuns: The image operator $\_ \text{`} \_$ on sets, e.g., has no analogue on FinFun sets.

Clearly, FinFuns are a very restricted set of functions, but we have demonstrated that this lightweight formalisation is in fact useful and easy to use. In Sec. 3, we have outlined the way to executing equality on FinFuns, but we need not stop there: Other operators like e.g. currying, λ-abstraction for FinFuns $'a \Rightarrow_f {}'b$ with $'a$ finite, and even the definite description operator $\iota x.\ \hat{P}_f\ x$ can all be made executable via the code generator. In terms of usability, FinFuns currently provide little support for defining new operators that can not be expressed by the existing ones: For example, recursive equations for the code generator must be stated explicitly, even if the definition explicitly uses the recursion combinator. But with some implementation effort, definitions and the code generator setup could be automated in the future.

For *quickcheck*, our implementation with at most quadratic complexity is sufficiently efficient because random FinFuns involve only a few updates. For larger applications, however, one is interested in more efficient representations. If, e.g., the domain of a FinFun is totally ordered, binary search trees are a natural option, but this requires considerable amount of work: (Balanced) binary trees must be formalised and proven correct, which could be based e.g. on [15], and all the operators that are recursive on a FinFun must be reimplemented. In practice, the user should not care about which implementation the code generator chooses, but such automation must overcome some technical restrictions, such as only one type variable for type classes or only unconditional rewrite rules for the code generator, perhaps by recurring on ad-hoc translations.

References

1. Berghofer, S., Nipkow, T.: Random testing in Isabelle/HOL. In: Proc. SEFM'04, pp. 230–239. IEEE Computer Society (2004)
2. Berghofer, S., Wenzel, M.: Inductive datatypes in HOL – lessons learned in formal-logic engineering. In: TPHOLs'99. LNCS, vol. 1690, pp. 19–36. Springer (1999)
3. Berghofer, S., Nipkow, T.: Executing higher order logic. In: TYPES'00. LNCS, vol. 2277, pp. 24–40. Springer (2002)
4. Collins, G., Syme, D.: A theory of finite maps. In: TPHOLs'95. LNCS, vol. 971, pp. 122–137. Springer (1995)
5. Dybjer, P., Haiyan, Q., Takeyama, M.: Combining testing and proving in dependent type theory. In: TPHOLs'03. LNCS, vol. 2758, pp. 188–203. Springer (2003)
6. Haftmann, F., Nipkow, T.: A code generator framework for Isabelle/HOL. Technical Report 364/07, Dept. of Computer Science, University of Kaiserslautern (2007)
7. Haftmann, F., Wenzel, M.: Constructive type classes in Isabelle. In: TYPES'06. LNCS, vol. 4502. Springer (2007)
8. Harrison, J.: Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-053, SRI International Cambridge Computer Science Research Centre (1995)
9. Klein, G., Nipkow, T.: A machine-checked model for a Java-like language, virtual machine and compiler. ACM TOPLAS 28, 619–695 (2006)
10. Krauss, A.: Partial recursive functions in higher-order logic. In: IJCAR'06. LNCS, vol. 4130, pp. 589–603. Springer (2006)
11. Lochbihler, A.: Jinja with threads. In: The Archive of Formal Proofs. `http://afp.sf.net/entries/JinjaThreads.shtml` (2007) Formal proof development.
12. Lochbihler, A.: Type safe nondeterminism - a formal semantics of Java threads. In: FOOL'08. (2008)
13. Lochbihler, A.: Code generation for functions as data. In: The Archive of Formal Proofs. `http://afp.sf.net/entries/FinFun.shtml` (2009) Formal proof development.
14. Nipkow, T., Paulson, L.C.: Proof pearl: Defining functions over finite sets. In: TPHOLs'05. LNCS, vol. 3603, pp. 385–396. Springer (2005)
15. Nipkow, T., Pusch, C.: AVL trees. In: The Archive of Formal Proofs. `http://afp.sf.net/entries/AVL-Trees.shtml` (2004) Formal proof development.
16. Urban, C.: Nominal techniques in Isabelle/HOL. Journal of Automatic Reasoning 40(4), 327–356 (2008)
17. Urban, C., Berghofer, S.: A recursion combinator for nominal datatypes implemented in Isabelle/HOL. In: IJCAR'06. LNCS, vol. 4130, pp. 498–512. Springer (2006)

A Notation

Isabelle/HOL formulae and propositions are close to standard mathematical notation. This subsection introduces non-standard notation, a few basic data types and their primitive operations.

Types is the set of all types which contains, in particular, the type of truth values *bool*, natural numbers *nat*, integers *int*, and the singleton type *unit* with its only element (). The space of total functions is denoted by $'a \Rightarrow 'b$. Type variables are written $'a$, $'b$, etc. The notation $t{::}\tau$ means that the HOL term t has type τ.

Pairs come with two projection functions *fst* and *snd*. Tuples are identified with pairs nested to the right: (a, b, c) is identical to $(a, (b, c))$ and $'a \times 'b \times 'c$ to $'a \times ('b \times 'c)$. Dually, the disjoint union of $'a$ and $'b$ is written $'a + 'b$.

Sets are represented as predicates (type $'a\ set$ is shorthand for $'a \Rightarrow bool$), but follow the usual mathematical conventions. $UNIV :: 'a\ set$ is the set of all elements of type $'a$. The image operator $f\ `\ A$ applies the function $f$ to every element of $A$, i.e. $f\ `\ A \equiv \{y \mid \exists x{\in}A.\ y = f\ x\}$. The predicate *finite* on sets characterises all finite sets. $card\ A$ denotes the cardinality of the finite set A, or 0 if A is infinite. $fold\ f\ z\ A$ folds a left-commutative[3] function $f{::}'a \Rightarrow 'b \Rightarrow 'b$ over a finite set $A :: 'a\ set$ with initial value $z{::}'b$.

Lists (type $'a\ list$) come with the empty list [] and the infix constructor $\_\cdot\_$ for consing. Variable names ending in "s" usually stand for lists and $|xs|$ is the length of *xs*. The function *set* converts a list to the set of its elements.

Function update is defined as follows: Let $f{::}'a \Rightarrow 'b$, $a{::}'a$ and $b{::}'b$. Then $f(a := b) \equiv \lambda x.\ \mathit{if}\ x = a\ \mathit{then}\ b\ \mathit{else}\ f\ x$.

The **option** data type $'a\ option$ adjoins a new element *None* to a type $'a$. All existing elements in type $'a$ are also in $'a\ option$, but are prefixed by *Some*. For succinctness, we write $\lfloor a \rfloor$ for $Some\ a$. Hence, for example, *bool option* has the values *None*, $\lfloor True \rfloor$ and $\lfloor False \rfloor$.

Partial functions are modelled as functions of type $'a \Rightarrow 'b\ option$ where *None* represents undefined and $f\ x = \lfloor y \rfloor$ means x is mapped to y. Instead of $'a \Rightarrow 'b\ option$, we write $'a \rightharpoonup 'b$ and call such functions **maps**. $f(x \mapsto y)$ is shorthand for $f(x := \lfloor y \rfloor)$. The domain of f (written $dom\ f$) is the set of points at which f is defined, $ran\ f$ denotes the range of f. The function $map\text{-}default\ b\ f$ takes a partial function f and continues it at its undefined points with b.

The **definite description** $\iota x.\ Q\ x$ is known as Russell's ι-operator. It denotes the unique x such that $Q\ x$ holds, provided exactly one exists.

[3] f is left-commutative, if it satisfies $f\ x\ (f\ y\ z) = f\ y\ (f\ x\ z)$ for all x, y, and z.