Light-weight containers for Isabelle: efficient, extensible, nestable

Andreas Lochbihler

Institute of Information Security, ETH Zurich
andreas.lochbihler@inf.ethz.ch

Abstract. In Isabelle/HOL, we develop an approach to efficiently implement container types such as sets and maps in generated code. Thanks to type classes and refinement during code generation, our light-weight framework is flexible, extensible, and easy to use. To support arbitrary nesting of containers, we devise an efficient linear order on sets that can even compare complements and non-complements. Our evaluation shows that it is both efficient and usable.

1 Introduction

Recently, executable implementations have been generated from increasingly large developments in theorem provers. Early works [3,8,16] implemented containers inefficiently, in particular as lists and closures, or burdened the formalisation with complex data structure details. Today, refinement approaches [5,6,9,10] enable the verification to operate on abstract types like sets and functions – for code generation, the refinement replaces them with efficient implementations. For use in large-scale projects, they should meet four requirements:

ease of use It requires little effort to apply the refinement to an application.
flexibility Applications themselves can choose which implementations to use for which container, and can easily switch between them; multiple implementations for the same container type are supported simultaneously.
extensibility When a user adds another implementation or a new type of stored data, he need not touch the existing parts. This is crucial for modularisation: Libraries can be included unchanged and extended incrementally.
nesting Containers can be nested arbitrarily, e.g., a map from sets to sets of sets.

For the proof assistant Isabelle/HOL and its code generator [7], the existing approaches differ in where the refinement happens. On the one hand, the Isabelle Collections Framework (ICF) [9] explicitly models refinement inside the logic: it defines a uniform interface to various verified data structures. It meets the above criteria except for ease of use and nesting. First, users must manually introduce copies for all definitions such that they use the interface and prove refinement subject to invariants of the data structure. In practice, refinement can make up a substantial part of the development [15]. Second, Isabelle's proof automation assumes unique representation of objects, but ICF-style refinement introduces multiple ones, e.g., both lists $[a, b]$ and $[b, a]$ implement the same set

$\{ a, b \}$. So, at present, the ICF does not support a set of sets implemented as unordered lists or red-black trees (RBT) – although demand for nested sets has been expressed on the Isabelle mailing list, e.g., posts 4D779F63.9050506@irit.fr and 89A2F83C-2AE8-44B7-B1ED-5EDD4E160C1F@loria.fr.

On the other hand, Haftmann et al. [6] advocate for refinement inside the code generator, see §1.1 for an introduction. This is easy to use, as the user need not bother about the details of implementations. In particular, representations remain unique in the logic, so nesting is possible. Yet, the current state in Isabelle2013 is far from fully exploiting the available features. For example, it supports only one implementation for each type, i.e., *all* sets are implemented either as lists or as RBTs. In a case study on extracting a Java interpreter [15], the elements of *some* sets could not be ordered linearly (as required for RBTs), so *all* sets were inefficiently implemented as lists.

Contributions We present an easy to use approach (called LC for light-weight containers) based on Haftmann's [6]. It supports multiple implementations and is flexibile, extensible, and nestable (§2). We devise a configurable scheme to automatically choose a suitable implementation based on the type of what is to be stored (§2.3). To avoid type class restrictions, we adapt a Haskell approach with type constructor classes [17] to Isabelle's single-parameter type classes. For the presentation, we focus on sets implemented as closures, lists, distinct lists, or RBTs. As our development covers all set operations from Isabelle, it can replace the default setup for code generation in applications without much ado. We have also covered maps, i.e., support the two fundamental container types.

Second, to support arbitrary nesting of sets, we devise an efficient linear order on sets (§3). It requires only linearly many comparisions between the elements and supports comparisons even between complements and non-complements. As Isabelle's automated disprover `quickcheck` [2] relies on code generation, it is important to support complements, too.

Third, we evaluate our approach in two micro-benchmarks and a case study (§4). The benchmarks show that our approach generates code as efficient as the ICF and that the linear order on sets is also efficient. The case study with the Java interpreter shows that our approach integrates seamlessly with the existing Isabelle setup and is therefore as easy to use; switching from Isabelle's default setup to ours did not require any adaptations. Moreover, multiple implementations for sets and maps improve the execution times.

Moreover, our way of combining the powerful features of the code generator is novel; we describe the ideas in §5. We would like to stress that we have proven all lemmas and refinements in this paper in Isabelle; LC is available online [14].

1.1 Background: the code generator framework and refinement

Isabelle's code generator [6,7] turns a set of equational theorems into a functional program with the same equational rewrite system. The translation guarantees partial correctness by construction, as it builds on equational logic.

Program refinement separates code generation issues from the rest of the formalisation in Isabelle. As any (executable) equational theorem suffices for code generation, the user may *locally* derive new (code) equations to use upon code generation. Hence, existing definitions and proofs remain unaffected.

For *data refinement*, the user replaces constructors of a datatype by other constants and derives equations that pattern-match on these new (pseudo-)constructors. Neither need the new constructors be injective and pairwise disjoint, nor exhaust the type. Again, this is local as it affects only code generation, but not the logical properties of the refined type. Thus, one cannot exploit the type's new structure inside the logic.

For example, Isabelle's default code generator setup represents sets (type α *set*) with the two pseudo-constructors *set* and *coset* of type $\alpha\ \mathit{list} \Rightarrow \alpha\ \mathit{set}$, which represent the (complement of the) finite set of elements in the given list. Note that they are neither injective ($\mathit{set}\ [1,2] = \mathit{set}\ [2,1]$, but $[1,2] \neq [2,1]$), nor do they exhaust the type α *set* if α is infinite (e.g., $\{1,3,5,\ldots\}$), nor are they disjoint if α is finite ($\mathit{set}\ [\mathit{True}] = \mathit{coset}\ [\mathit{False}]$ for booleans). Nevertheless, the following equations implement the membership test $\in$ on type α *set*:

$$(x \in \mathit{set}\ xs) = (\mathit{memb}\ xs\ x) \qquad\qquad (x \in \mathit{coset}\ xs) = (\neg\, \mathit{memb}\ xs\ x) \tag{1}$$

where $\mathit{memb}\ xs\ x$ checks if x equals one of xs's elements by traversing xs – the type class *equal* provides the implementation of these equality tests.

This is an example of *sort refinement*: The equality test in *memb*'s code equations requires that α is of sort *equal*, but *memb*'s specification in the logic as $\lambda xs\ x.\ x \in \mathit{set}\ xs$ does not. The code generator collects, propagates, and checks all sort constraints upon code generation. Thus, it propagates $\alpha :: \mathit{equal}$ via (1) to $\in$, too. Sorts intersect: Suppose that another pseudo-constructor *tree* represents finite sets as binary search trees and we use $(x \in \mathit{tree}\ t) = (\mathit{lookup}\ t\ x \neq \mathit{None})$, where *lookup* obtains the linear order on α from the type class *linorder*. Then, the code generator enforces that any invocation of $\in$ operates on sets whose element type instantiates both *equal* and *linorder*.

1.2 Related work

The ICF approach [9] considers refinement inside the logic superior to refinement in the code generator, as the implementation can exploit the refined structure and, e.g., resolve non-determinism from underspecification such as the order of iteration over a set. However, the ICF requires more adaptations; some automation has been developed for monadic programs [10], but this still requires adapting the application to the refinement calculus. We argue that both approaches complement each other, and recommend to use the ICF only when necessary, and to stick with the simpler refinement in the code generator whenever possible. For example, in [15], we used Haftmann's approach when sets and maps are accessed only through membership tests and lookup operations, respectively, and the ICF to resolve non-determinism, e.g., to pick an arbitrary element from a set.

Peyton Jones [17] and Chen et al. [4] show that Haskell's single-parameter type classes do not suffice for bulk-type polymorphism (flexibility and extensibility in our terminology). Our approach nevertheless succeeds, because refinement is incremental and we do not have to extend the generated code itself.

Lescuyer's Containers library [11] for Coq efficiently implements finite sets and maps with type classes, but he cannot represent set complements. His linear order on sets is based on the lexicographic ordering of the sorted list of elements.

2 Multiple implementations for containers

Sort intersection as described in §1.1 creates problems in large-scale applications that use the same HOL container type such as α *set* for different α, as some element types α fail to meet all sort constraints. In previous work [15], e.g., some sets contain strings (a sequence of characters) and others functions. Unfortunately, we could not make strings an instance of *linorder*, because the order on lists had already been fixed to the partial prefix order elsewhere, and functions lack computable equality for the type class *equal*. Thus, we had to stick with inefficient lists which allow duplicates, see §4.3 for details.

In this section, we introduce new type classes which *any* type can be made an instance of and show how to support multiple implementations (§2.1). Thus, sort intersection is no longer a show-stopper. Then, we demonstrate extensibility by adding new data and a new implementation (§2.2). To improve usability, we devise a configurable scheme for automatically choosing an implementation (§2.3) and show how to deal with binary operations (§2.4).

2.1 New type classes and multiple implementations

To avoid instantiation obstacles, Peyton Jones [17, §3] introduces new type classes whose parameters already tell whether the operation is supported. Here, we borrow his idea and adapt it to the theorem prover setting. We introduce a new type class *ceq* for equality on container elements (to make the overloading explicit, we write the type parameter as a superscript to type class parameters):

$$\begin{aligned}\textbf{class}\ \mathit{ceq} = \text{fixes}\ &\mathit{ceq}^{\alpha} :: (\alpha \Rightarrow \alpha \Rightarrow \mathit{bool})\ \mathit{option}\\ \text{assumes}\ &\mathit{ceq}^{\alpha} = \lfloor \mathit{eq} \rfloor \Longrightarrow \mathit{eq} = (\mathit{op} =)\end{aligned}$$

The declared equality operator ceq^{α} tells whether elements of type α may be tested for equality at all. If so ($\mathit{ceq}^{\alpha} = \lfloor \mathit{eq} \rfloor$ for some *eq*; $\lfloor \_ \rfloor$ denotes definedness), the assumption enforces that *eq* in fact implements HOL equality $\mathit{op} =$ (we forbid other congruence relations, as Isabelle's proof automation cannot handle them well). Otherwise ($\mathit{ceq}^{\alpha} = \mathit{None}$), *ceq* does not impose any constraints on the implementation and – as all proofs rely only on the specified assumptions – neither must any usage of ceq^{α}. Thus, *every* type can be made an instance of *ceq*. For example, the instantiations for the function space constructor *fun* (written infix as $\Rightarrow$) and natural numbers *nat* are as follows:

instantiation *fun* :: (*type*, *type*) *ceq* begin
definition $\mathit{ceq}^{\alpha \Rightarrow \beta} = \mathit{None}$
instance ⟨⟨proof⟩⟩ end

instantiation *nat* :: *ceq* begin
definition $\mathit{ceq}^{\mathit{nat}} = \lfloor \mathit{op} = \rfloor$
instance ⟨⟨proof⟩⟩ end

Unfortunately, we cannot follow Peyton Jones' development any further: he introduces a type constructor class for collection type constructors that specifies the operations, but Isabelle does not support type constructor classes. Instead, we use data refinement. For the moment, we only consider three implementations: lists with and without duplicates and characteristic functions; in §2.2, we add efficient RBTs. To that end, we define four pseudo-constructors for α *set* that replace *set* and *coset* for the code generator:

char. function	*ChF*	$:: (\alpha \Rightarrow \mathit{bool}) \Rightarrow \alpha\ \mathit{set}$	$\mathit{ChF}\ P$	$= \{\, x.\ P\ x \,\}$
monad style	*MSet*	$:: \alpha\ \mathit{list} \Rightarrow \alpha\ \mathit{set}$	$\mathit{MSet}\ xs$	$= \mathit{set}\ xs$
distinct list	*DSet*	$:: \alpha\ \mathit{dlist} \Rightarrow \alpha\ \mathit{set}$	$\mathit{DSet}\ ds$	$= \{\, x.\ \mathit{dmemb}\ ds\ x \,\}$
complement	*Compl*	$:: \alpha\ \mathit{set} \Rightarrow \alpha\ \mathit{set}$	$\mathit{Compl}\ A$	$= \overline{A}$

For α of sort *ceq*, the type α *dlist* consists of all lists from α *list* whose elements are pairwise distinct w.r.t. the equality operator from *ceq*; if ceq^{α} is undefined, α *dlist* consists of all lists. *dmemb ds x* checks if x occurs in *ds* using *ceq*'s equality operator. We model the complement of a set A (notation $\overline{A}$) as *Compl A*.

Monad-style sets (*MSet*) can be used to model non-determinism. They allow duplicates and avoid equality checks whenever possible, e.g., for *insert*, $\cup$, and *bind*. Still, we do implement operations that require equality, but they may fail at run time when ceq^{α} is *None*. Membership, e.g., uses the following equation:

$$(x \in \mathit{MSet}\ xs) = (\mathit{case}\ \mathit{ceq}^{\alpha}\ \mathit{of}\ \mathit{None} \Rightarrow \mathit{error}\ (\lambda\_.\ x \in \mathit{MSet}\ xs) \\ \mid \lfloor eq \rfloor \Rightarrow \mathit{memb'}\ eq\ xs\ x) \tag{2}$$

where *error* logically returns its argument applied to the unit value (), but it raises an exception in the generated code at run time; the unit closure ensures termination in call-by-value languages like ML. As *memb'* takes the equality operation as a parameter, we do not depend on the type class *equal*. Note that membership $\in$ cannot be total, as we deliberately do not require an implementation for equality. Like in the common case of missing patterns in functional programs, the user himself must ensure that ceq^{α} is defined for the element type α whenever he calls $\in$ on *MSet*. The other pseudo-constructors do not need such a run time check, as we have defined them logically in terms of membership:

$$(x \in \mathit{ChF}\ P) = P\ x \qquad (x \in \mathit{DSet}\ ds) = \mathit{dmemb}\ ds\ x \qquad (x \in \mathit{Compl}\ A) = (x \notin A)$$

2.2 Extensibility

Extensibility expresses that one may use containers with new types of elements and add new implementations for a container without editing the existing code base. This ensures that users can freely extend a container framework. In this section, we demonstrate that our light-weight approach achieves this.

To use a new type of elements, one merely has to instantiate the new type classes, i.e., *ceq* in the example. As the operations can default to *None*, this is always possible. For example, arithmetic expressions:

datatype $\mathit{expr} = \mathit{Val}\ \mathit{nat} \mid \mathit{Var}\ \mathit{string} \mid \mathit{Plus}\ \mathit{expr}\ \mathit{expr} \mid \mathit{Times}\ \mathit{expr}\ \mathit{expr}$
instantiation *expr* :: *ceq* begin
definition $\mathit{ceq}^{\mathit{expr}} = \lfloor op = \rfloor$
instance ⟨⟨proof⟩⟩ end

Note that the datatype declaration already generates code equations for *op* =.

Next, we show how to add an implementation of sets backed by RBTs. This requires a linear order on container elements, i.e., a new type class *corder* (where *class.linorder leq lt* denotes that *leq* is a linear order and *lt* its strict version):

$$\begin{aligned}&\textbf{class}\ \mathit{corder} = \text{fixes}\ \mathit{corder}^{\alpha} :: ((\alpha \Rightarrow \alpha \Rightarrow \mathit{bool}) \times (\alpha \Rightarrow \alpha \Rightarrow \mathit{bool}))\ \mathit{option}\\ &\qquad\qquad\qquad \text{assumes}\ \mathit{corder}^{\alpha} = \lfloor(\mathit{leq}, \mathit{lt})\rfloor \Longrightarrow \mathit{class.linorder}\ \mathit{leq}\ \mathit{lt}\end{aligned}$$

Our RBT implementation builds on the verified RBT formalisation in Isabelle's library, which is based on the *linorder* type class; we only have to adapt it to *corder*. Analogous to α *dlist*, the type α *srbt* of RBT sets contains all RBTs that are sorted according to *corder*, if *corder* is defined; otherwise, it contains all binary trees irrespective of sorting and balancing. Then, we define the new pseudo-constructor $\mathit{RSet}\ \mathit{rs} = \{\, x.\ \mathit{rmemb}\ \mathit{rs}\ x \,\}$ where *rmemb* denotes the lookup operation on α *srbt*, which uses *corder* for comparing elements. To finish, we declare *RSet* as another pseudo-constructor for α *set* and prove code equations for all set operations. Again, operations like *insert* are correctly implemented only if *corder* implements some linear order; otherwise, they fail with an exception during execution.

$$\begin{aligned}(x \in \mathit{RSet}\ \mathit{rs}) &= \mathit{rmemb}\ \mathit{rs}\ x\\ \mathit{insert}\ x\ (\mathit{RSet}\ \mathit{rs}) &= (\mathit{case}\ \mathit{corder}^{\alpha}\ \mathit{of}\ \mathit{None} \Rightarrow \mathit{error}\ (\lambda\_.\ \mathit{insert}\ x\ (\mathit{RSet}\ \mathit{rs}))\\ &\qquad\qquad\qquad\qquad\quad \mid \lfloor\_\rfloor \Rightarrow \mathit{RSet}\ (\mathit{rinsert}\ x\ \mathit{rs}))\end{aligned} \tag{3}$$

Sort refinement requires that from now on all element types inhabit *corder*, too, as α has sort *corder* in (3). Hence, we instantiate *corder* for the element types. Since it is very similar to the *linorder* type class, the instantiations are canonical and full Isabelle support is available, e.g., Thiemann's order generator for data types [19]. For example, the proofs in the following are automatic.

instantiation *nat* :: *corder* begin
definition $\mathit{ceq}^{\mathit{nat}} = \lfloor(\mathit{op} \leq, \mathit{op} <)\rfloor$
instance ⟪proof⟫ end

derive *linorder expr*
instantiation *expr* :: *corder* begin
definition $\mathit{ceq}^{\mathit{expr}} = \lfloor(\mathit{op} \leq, \mathit{op} <)\rfloor$
instance ⟪proof⟫ end

2.3 Automatically choosing an implementation

Recall from (2) and (3) that the generated code raises a run-time exception in case of an unsupported operation. One can analyse the code equations to that end before code generation, but we have not yet implemented such an analysis. Instead, we let the generated code choose the implementation based on the operations the element type provides, e.g., use *RSet* only if corder^{α} is defined. Then, we are sure that the necessary operations are available, i.e., the exception in (3) cannot occur. However, exceptions can still occur when a set operation has no implementation at all. For a set of functions, e.g., $\in$ will still fail in (2). This is a design choice: We want to support α *set* for *all* α. If α does not permit to implement a set operation at all, it is the user's fault to apply the operation to α *set*.

Peyton Jones [17, §3.3] proposes a similar approach, but his is neither extensible nor flexible. He avoids the implementability problem by requiring equality for all element types, but this does not fit to how sets are used in Isabelle.

All sets originate from either a set comprehension $\mathit{ChF}\ P$ or the empty set $\emptyset$. We ignore the pseudo-constructor *ChF*, as the operations for *ChF* do not depend on *ceq* or *corder*. For $\emptyset$, our code equations heuristically pick an implementation. The following is a first, naïve attempt in the style of [17]:

$$\begin{aligned}\emptyset = (&\mathit{case}\ \mathit{corder}^{\alpha}\ \mathit{of}\ \lfloor \_ \rfloor \Rightarrow \mathit{RSet}\ \mathit{rempty}\\ &\mid \mathit{None} \Rightarrow \mathit{case}\ \mathit{ceq}^{\alpha}\ \mathit{of}\ \lfloor \_ \rfloor \Rightarrow \mathit{DSet}\ \mathit{dempty} \mid \mathit{None} \Rightarrow \mathit{MSet}\ [\,])\end{aligned} \tag{4}$$

This equation uses RBTs if there is a linear order on the elements. Otherwise, it tries distinct lists for equality and picks monad-style sets as last resort. This way, RBTs are only used for element types that do provide a linear order *corder*. Thus, the error in (3) cannot trigger. Nevertheless, we cannot eliminate the check, as the user still can misuse *RSet* in his own equations. However, (4) offers too little control over the choice and thus violates flexibility. For some types, it is sensible to use distinct lists even if there is a linear order – for *bool* with just two elements, e.g., the *DSet* implementation is four times faster than the *RSet* one.[1]

Instead, we let an overloaded operation choose the implementation:

$$\textbf{class}\ \mathit{set\text{-}impl} = \text{fixes}\ \mathit{set\text{-}impl}^{\alpha} :: \mathit{set\text{-}impl}$$

The type *set-impl* (inhabited by only one value *Set-IMPL*) has a pseudo-constructor for each implementation: *Set-ChF*, *Set-dlist*, *Set-RBT*, *Set-Monad*, plus *Set-Auto* for automatic selection like in (4). They are only pseudo-constructors such that we can add more for new implementations later. Then, we implement $\emptyset$ via $\emptyset = \mathit{sempty}\ \mathit{set\text{-}impl}^{\alpha}$, where the function *sempty*, logically defined by $\mathit{sempty}\ \mathit{Set\text{-}IMPL} = \emptyset$, chooses the desired implementation:

$$\begin{array}{ll} \mathit{sempty}\ \mathit{Set\text{-}ChF} = \mathit{ChF}\ (\lambda\_.\ \mathit{False}) & \mathit{sempty}\ \mathit{Set\text{-}dlist} = \mathit{DSet}\ \mathit{dempty}\\ \mathit{sempty}\ \mathit{Set\text{-}RBT} = \mathit{RSet}\ \mathit{rempty} & \mathit{sempty}\ \mathit{Set\text{-}Monad} = \mathit{MSet}\ [\,]\\ \mathit{sempty}\ \mathit{Set\text{-}Auto} = (\mathit{case}\ \mathit{corder}^{\alpha}\ \mathit{of}\ \lfloor \_ \rfloor \Rightarrow \mathit{RSet}\ \mathit{rempty}\ \mid\ \ldots) & \end{array}$$

Note that we could have overloaded $\emptyset$ directly without the detour *set-impl* and *sempty*. Yet, as Isabelle allows overloading only for constants with exactly one type parameter, this does not extend to other container types like maps with multiple type parameters. Our approach also works such container types.

Below, we give three example instantiations for *set-impl*. As motivated above, *bool* chooses distinct lists although $\mathit{corder}^{\mathit{bool}}$ is defined. α *option* (the type of $\lfloor \_ \rfloor$ and *None*) inherits the choice from its type argument, as it adds only one value. In contrast, a set of sets discards any preference from the element type and falls back on automatic selection. This seems sensible, as a set of sets can become much larger than a set of the elements.

$$\mathit{set\text{-}impl}^{\mathit{bool}} = \mathit{Set\text{-}dlist} \qquad \mathit{set\text{-}impl}^{\alpha\ \mathit{option}} = \mathit{set\text{-}impl}^{\alpha} \qquad \mathit{set\text{-}impl}^{\alpha\ \mathit{set}} = \mathit{Set\text{-}Auto}$$

[1] Build, e.g., the set $\{\mathit{True}, \mathit{False}\}$ and check membership for both elements. Under PolyML, 1M (10M) iterations take .05 s (.47 s) for *DSet* and .21 s (2.05 s) for *RSet*.

Moreover, users can later change the implementations for a type α, if they want to: as all pseudo-constructors are logically equivalent, proving a different code equation for $\mathit{set\text{-}impl}^{\alpha}$ is straightforward: As all pseudo-constructors are logically equal to $\mathit{Set\text{-}IMPL}$, we can, e.g., prove $\mathit{set\text{-}impl}^{\alpha\ \mathit{option}} = \mathit{Set\text{-}ChF}$ and use this code equation to choose characteristic functions as default for α *option*.

2.4 Binary operations

Binary operations like $\cap$ and $\cup$ require pattern-matching on both sets, i.e., the number of possible combinations grows quadratically with the number of implementations. In the ICF [9], a Ruby script automatically generates implementations for all combinations, all of which use a generic implementation parametrised by iterators and basic operations. More efficient implementations for special combinations (like intersecting two *RSet*s [1]) are not supported.

With our approach, sequential pattern matching in the target language offers a better solution. First, we derive general equations that pattern-match only on one constructor and compute the result generically. Second, we show equations for special cases with more efficient implementations, which take precedence over the generic ones as pattern matching is sequential. This keeps the number of equations linear in the number of implementations plus the optimised cases. Moreover, the general equations automatically cover future set implementations.

For intersection and *RSet*, e.g., we obtain the following, where $\mathit{rfilter}\ P\ rs$ retains only rs's elements that satisfy the predicate P and *rint* is the fast intersection algorithm on α *srbt*.

$$\begin{aligned}
&\mathit{RSet}\ rs_1 \cap \mathit{RSet}\ rs_2 = (\mathit{case}\ \mathit{corder}^{\alpha}\mathit{of}\ \mathit{None}\ \Rightarrow \ldots \mid \lfloor \_ \rfloor \Rightarrow \mathit{RSet}\ (\mathit{rint}\ rs_1\ rs_2))\\
&\mathit{RSet}\ rs \cap A = (\mathit{case}\ \mathit{corder}^{\alpha}\mathit{of}\ \mathit{None} \Rightarrow \ldots \mid \lfloor \_ \rfloor \Rightarrow \mathit{RSet}\ (\mathit{rfilter}\ (\lambda x.\ x \in A)\ rs))\\
&A \cap \mathit{RSet}\ rs = (\mathit{case}\ \mathit{corder}^{\alpha}\mathit{of}\ \mathit{None} \Rightarrow \ldots \mid \lfloor \_ \rfloor \Rightarrow \mathit{RSet}\ (\mathit{rfilter}\ (\lambda x.\ x \in A)\ rs))
\end{aligned}$$

When we prove these equations, we cannot exploit sequentiality of pattern matching, i.e., we implicitly prove that all right-hand sides are equal when the left-hand sides unify. This is only possible as refinement happens in the code generator, i.e., our pseudo-constructors abstract from the concrete representation. As the ICF models the refinement in the logic, it cannot prove such equalities.

3 Executable linear order on sets

Recall that our approach abstracts from different implementations in the logic. Thus, it directly supports arbitrary nesting of containers, provided that we make the container type an instance of the type classes – in our example, *ceq* and *corder* for α *set*. Not to lose on efficiency, we now devise a linear order $\sqsubseteq$ on sets and implement $\mathit{corder}^{\alpha\ \mathit{set}} = \lfloor(\sqsubseteq, \sqsubset)\rfloor$. This is one example where the separate type class *corder* is crucial: As Isabelle fixes the canonical order $\leq$ on α *set* to the non-linear subset order $\subseteq$, we cannot make α *set* an instance of *linorder*.

By the axiom of choice, there is a linear order on every set, but we cannot implement this order, so it is useless here. Fortunately, it suffices if we can decide the ordering on representable sets. Given a linear order $\leq$ on the elements, we first define a linear order on the finite and cofinite sets. Then, we extend it to a linear order on all sets by the axiom of choice (§3.1), as *corder* requires a linear order on all elements. Our equations for code generation (§3.2) pattern-match on the pseudo-constructors which represent only finite or cofinite sets. If the (co)finite sets are given as sorted lists of their (non-)elements, $\sqsubseteq$ requires at most linearly (in the size of the lists) many $\leq$-comparisons – except for comparing a finite and a cofinite set when α is finite, because a set may be both finite and cofinite. For the latter case, we show that further operations on α are necessary, and implement them for α *set* to ensure nestability and extensibility (§3.3).

Moreover, our order $\sqsubseteq$ satisfies the following properties for all sets A, A' and finite sets F, F', which facilitate proving the code equations in §3.2:

(P1) $\emptyset \sqsubseteq A$ and $A \sqsubseteq \mathit{UNIV}$
(P2) If $F \subseteq F'$, then $F \sqsubseteq F'$
(P3) $\overline{A} \sqsubseteq \overline{A'}$ iff $A' \sqsubseteq A$
(P4) If α is infinite, then $F \sqsubseteq \overline{F'}$

Properties P1 and P2 describe the similarity with the subset order: the empty set $\emptyset$ and the full set *UNIV* of all elements are the least and greatest sets, resp., and $\sqsubseteq$ extends $\subseteq$ on finite sets. Hence, when iterating over a set of finite sets in ascending order, one visits subsets before supersets. P3 allows to drop the *Compl* constructor on both sides if the relation is reversed, i.e., complement is anti-monotone. P4 expresses that finite sets are always less than cofinite sets, if α's universe is infinite. If α is finite, $F \sqsubseteq \overline{F'}$ is inconsistent with P2 ($\emptyset \subseteq \{\, a \,\} \sqsubseteq \overline{\mathit{UNIV}} = \emptyset$, i.e., $\{\, a \,\} = \emptyset$, a contradiction), because then sets are both finite and cofinite.

3.1 Definition

We construct our linear order $\sqsubseteq$ from two intermediate partial orders $\sqsubseteq_1$ and $\sqsubseteq_2$, where each step extends the previous order.

First, we define that $A \sqsubset_1 B$ holds whenever both A and B are finite and B contains the mininum element of the symmetric difference of A and B. Intuitively, if $A = \mathit{set}\ \mathit{as}$ and $B = \mathit{set}\ \mathit{bs}$ with *as* and *bs* duplicate-free and $\leq$-sorted in ascending order, $A \sqsubset_1 B$ iff *as* precedes *bs* in the lexicographic list order w.r.t. the converse $\geq$ of $\leq$ (Lem. 1). For a three-value type with order $0 < 1 < 2$, e.g., $\sqsubset_1$ orders the sets as follows:

$$\emptyset \sqsubset_1 \{\, 2 \,\} \sqsubset_1 \{\, 1 \,\} \sqsubset_1 \{\, 1,2 \,\} \sqsubset_1 \{\, 0 \,\} \sqsubset_1 \{\, 0,2 \,\} \sqsubset_1 \{\, 0,1 \,\} \sqsubset_1 \{\, 0,1,2 \,\} \qquad (5)$$

Taking the converse of $\leq$ is crucial for the above properties. In the example, the lexicographic list order w.r.t. $\leq$ would give $\{\, 0,1,2 \,\} \sqsubset_1 \{\, 1 \,\}$, which violates P1 and P2. Moreover, note the symmetry with complements in (5): the n-th set from the left is the complement of the n-th set from the right.

For finite types like in (5), $\sqsubset_1$ completely determines $\sqsubseteq$. So let us move on to infinite types. Fix a set of sets $\mathfrak{C} :: \alpha\ \mathit{set}\ \mathit{set}$ with two properties: (i) If $A :: \alpha\ \mathit{set}$ is finite, then $A \in \mathfrak{C}$. (ii) If α is infinite, then $A \in \mathfrak{C}$ iff $\overline{A} \notin \mathfrak{C}$. Such a $\mathfrak{C}$ exists: If α is finite, take *UNIV*. Otherwise, consider the subset order restricted to sets of sets that satisfy (i) and the "only if" direction of (ii). For any chain in this order, the union of the chain's sets of sets is an upper bound. Thus, by Zorn's lemma, the order has a maximal element – and all maximal elements satisfy (i) and (ii). For infinite α, the set $\mathfrak{C}$ decides for each set A if $\sqsubseteq$ treats it like a finite set ($A \in \mathfrak{C}$) or like a cofinite set ($A \notin \mathfrak{C}$). If A is infinite, this decision does not matter, as we do not care about infinite sets, but it ensures complement symmetry P3.

Next, $\sqsubset_2$ extends $\sqsubset_1$ to a linear order on $\mathfrak{C}$ such that $\emptyset$ is the least element. Hence, $\emptyset \sqsubseteq_2 A$ even if $A \in \mathfrak{C}$ is infinite. By the order extension principle, $\sqsubset_2$ exists. This suffices for our purpose, as we care only about finite sets, i.e., we need not specify $\sqsubset_2$ completely for infinite A and B.

Finally, we mirror $\sqsubset_2$ at the boundary of $\mathfrak{C}$ to obtain complement symmetry P3. We define $\sqsubset$ as follows:

$$A \sqsubset B = (\textit{if } A \in \mathfrak{C} \textit{ then } A \sqsubset_2 B \vee B \notin \mathfrak{C} \textit{ else } B \notin \mathfrak{C} \wedge \overline{B} \sqsubset_2 \overline{A})$$

Property P4 holds as the cofinite sets, which are not in $\mathfrak{C}$, are greater than the finite ones (which are members of $\mathfrak{C}$). Note that if α is finite, $\mathfrak{C} = \mathit{UNIV}$ and therefore, $\sqsubset$ is identical to $\sqsubset_2$ and $\sqsubset_1$.

3.2 Code equations

Now, we derive code equations to implement $\sqsubset$ and $\sqsubseteq$ for finite and cofinite sets. In the following, we assume that a (co)finite set is given as a sorted and duplicate-free list of (the complement's) elements. Thanks to the above properties, some combinations are straightforward: P3 reduces comparisons between two cofinite sets to comparing their complements; and P4 already decides comparisons between one finite and one cofinite set if α is infinite. Thus, only two cases are left: comparing two finite sets and – if α is finite – comparing a finite and a cofinite set.

For the first case, we show that $\sqsubset_1$ is a kind of lexicographic ordering:

Lemma 1. *Let A and B be non-empty, finite sets and let $\mathit{Min}\ A$ and $\mathit{Min}\ B$ denote their respective minimum elements. Then, $A \sqsubset_1 B$ iff $\mathit{Min}\ A > \mathit{Min}\ B$, or $\mathit{Min}\ A = \mathit{Min}\ B$ and $A - \{\mathit{Min}\ A\} \sqsubset_1 B - \{\mathit{Min}\ B\}$.*

Corollary 1. *Let xs and ys be sorted, duplicate-free lists. Then, set $xs \sqsubset_1$ set ys iff* lexord $(>)$ $xs\ ys$, *where* lexord $(>)$ *is the lexicographic order on lists for the element order $>$.*

Thus, we can implement comparisons between finite sets efficiently as a lexicographic order. If we store the sets in sorted order (like RBTs do), the number of element comparisons is linear in the size of the sets.

Now, only comparisons between a finite and a cofinite set remain if α is finite. Unfortunately, we cannot (computationally) decide such comparisons solely by looking at the representations of finite and cofinite sets. This is not a fault of our choice for $\sqsubset$, but impossible for any linear order $\Subset$ implemented as a polymorphic function of type $\alpha\ \mathit{set} \Rightarrow \alpha\ \mathit{set} \Rightarrow \mathit{bool}$. To see this, compare the sets $\overline{\{a\}}$ and $\emptyset$. If

a is the only value that inhabits α, $\overline{\{ a \}} = \emptyset$ and therefore $\overline{\{ a \}} \not\Subset \emptyset$ and $\emptyset \not\Subset \overline{\{ a \}}$, as $\Subset$ is irreflexive. Otherwise, $\overline{\{ a \}} \neq \emptyset$ and thus $\overline{\{ a \}} \Subset \emptyset$ or $\emptyset \Subset \overline{\{ a \}}$ by linearity. Thus, $\Subset$ must know whether α contains further values, but it cannot compute that solely from its arguments $\{ a \}$ and $\overline{\emptyset}$. Similar examples show that $\Subset$ has to know if there are further values above, below, or between any two values.

Therefore, we introduce another overloaded operation *proper interval* pi^{α} of type α *option* $\Rightarrow \alpha$ *option* $\Rightarrow$ *bool* that checks whether an open interval is proper, i.e., non-empty. Intervals are given by their bounds, *None* represents unboundedness. Hence, all implementations of pi^{α} satisfy the following specification (note that $(\exists z.\ True) = True$ as all HOL types are inhabited):

$$\begin{array}{ll} pi^{\alpha}\ None\ None = True & pi^{\alpha}\ None\ \lfloor y \rfloor = (\exists z.\ z < y) \\ pi^{\alpha}\ \ \lfloor x \rfloor\ \ None = (\exists z.\ x < z) & pi^{\alpha}\ \ \lfloor x \rfloor\ \ \lfloor y \rfloor = (\exists z.\ x < z \wedge z < y) \end{array} \tag{6}$$

Now, we present the case of comparing a complement with a non-complement. To decide $\overline{A} \sqsubseteq_1 B$, where $A = set\ xs$ and $B = set\ ys$ are given by sorted and duplicate-free lists, we use the following function *cle* of type α *option* $\Rightarrow \alpha$ *list* $\Rightarrow$ α *list* $\Rightarrow$ *bool* – a similar function *lec* (not shown) deals with other case $A \sqsubseteq_1 \overline{B}$:

$$\begin{array}{llll} cle\ b & [] & [] & = \neg\, pi^{\alpha}\ b\ None \\ cle\ b & (x \cdot xs) & [] & = \neg\, pi^{\alpha}\ b\ \lfloor x \rfloor \wedge cle\ \lfloor x \rfloor\ xs\ [] \\ cle\ b & [] & (y \cdot ys) & = \neg\, pi^{\alpha}\ b\ \lfloor y \rfloor \wedge cle\ \lfloor y \rfloor\ []\ ys \\ cle\ b & (x \cdot xs) & (y \cdot ys) & = (if\ x < y\ then\ \neg\, pi^{\alpha}\ b\ \lfloor x \rfloor \wedge cle\ \lfloor x \rfloor\ xs\ (y \cdot ys) \\ & & & \quad\ else\ if\ y < x\ then\ \neg\, pi^{\alpha}\ b\ \lfloor y \rfloor \wedge cle\ \lfloor y \rfloor\ (x \cdot xs)\ ys \\ & & & \quad\ else\ \neg\, pi^{\alpha}\ b\ \lfloor x \rfloor) \end{array} \tag{7}$$

The additional parameter b acts as a lower bound: *cle* b interprets the complement $\overline{set\ xs}$ with respect to the set of values greater than b (notation $b\!\uparrow$) instead of *UNIV*. As it further assumes $set\ xs \cup set\ ys \subseteq b\!\uparrow$, it ignores all values not in $b\!\uparrow$. For example, *cle* $\lfloor 0 \rfloor$ treats the type from (5) as if it were $1 < 2$, i.e., it considers only the left half of (5).

Lemma 2. *Let α be finite, and xs and ys be sorted and duplicate-free, and $set\ xs \cup set\ ys \subseteq b\!\uparrow$. Then, $\overline{set\ xs} \cap b\!\uparrow \sqsubseteq set\ ys$ iff $cle\ b\ xs\ ys$.*

Corollary 2. *If α is finite, xs and ys are sorted and duplicate-free, then*

$$\overline{set\ xs} \sqsubseteq set\ ys = cle\ None\ xs\ ys.$$

Let us see how *cle* works. The first two cases correspond to $\overline{A} \cap b\!\uparrow \sqsubseteq \emptyset$ for $A = set\ []$ or $A = set\ (x \cdot xs)$. By P1, this holds iff $\overline{A} \cap b\!\uparrow = \emptyset$ – and the two equations use pi^{α} to test if A exhausts $b\!\uparrow$. The third case is symmetric: $b\!\uparrow \sqsubseteq set\ (y \cdot ys)$ holds iff $b\!\uparrow \subseteq set\ (y \cdot ys)$, i.e., $set\ (y \cdot ys)$ exhausts $b\!\uparrow$. The last case is the most interesting one. Suppose that $x < y$. Then, there must not be a value between b and x; otherwise ($pi^{\alpha}\ b\ \lfloor x \rfloor$), the minimum element of $\overline{A} \cap b\!\uparrow$ is lower than the minimum element y of B, so $\overline{A} \cap b\!\uparrow \sqsupset_1 B$ by Lem. 1. Moreover, neither $y \cdot ys$ nor the complement of $x \cdot xs$ contain x, as the lists are sorted and duplicate-free. Thus, $\overline{set\ (x \cdot xs)} \cap b\!\uparrow = \overline{set\ xs} \cap \lfloor x \rfloor\!\uparrow$ and *cle* recurses. Now suppose that $y < x$. Then, there must not be a value between b and y; otherwise $\mathit{Min}\ (\overline{A} \cap b{\uparrow}) < \mathit{Min}\ B$ and thus $B \sqsubset_1 \overline{A} \cap b{\uparrow}$ by Lem. 1. As $y < x$ and the lists are sorted, y is the minimum element of both $\overline{A} \cap b{\uparrow}$ and B. Thus, we can remove y from both sets in the recursive call by raising b to $\lfloor y \rfloor$ and dropping y from B. This is correct by Lem. 1. Finally, if $x = y$, we have found an element in which the sets differ. If $\neg\, \mathit{pi}\ b\ \lfloor x \rfloor$, then $y \in B$ is the minimum element of the symmetric difference between $\overline{A} \cap b{\uparrow}$ and B, so $B \sqsubset \overline{A} \cap b{\uparrow}$. Otherwise, the converse holds.

As can be seen from *cle*'s definition, every case requires at most two comparisons and one call to *pi*, and every recursive call consumes one list element. Thus, deciding $\overline{A} \sqsubseteq B$ is linear in the size of A and B if α is finite – for infinite α, this takes constant time by P4. To decide whether α is infinite, we use another type class to overload FIN^α with the meaning of *finite UNIV*. Finite types implement FIN^α as *True*, infinite ones as *False*.

In summary, (8) below implements the total order on sets, where ... represents the usual test for corder^α being defined and that A and B are finite (except for the first equation). The function *s2l A* returns A's elements as a sorted (w.r.t. corder^α) and duplicate-free list – for $A = \mathit{RSet}\ rs$, *s2l rs* traverses rs in-order; for *DSet* (*MSet*), *s2l* sorts the elements (and removes duplicates); it fails with an exception for *ChF* and *Compl* as expected. The element type α must instantiate the type classes *corder*, *pi* and *FIN*. Note how (8) exploits that pattern matching is sequential (cf. §2.4): the last equation, e.g., executes only if A and B are no complements. Thus, sequentiality saves us from manually implementing all 28 cases.

$$\begin{array}{lcll}
\mathit{Compl}\ A \sqsubseteq \mathit{Compl}\ B & = \dots & B \sqsubseteq A \\
\mathit{Compl}\ A \sqsubseteq \quad B & = \dots & \mathit{FIN}^\alpha \wedge \mathit{cle}\ \mathit{None}\ (\mathit{s2l}\ A)\ (\mathit{s2l}\ B) \\
\quad A \quad \sqsubseteq \mathit{Compl}\ B & = \dots & \mathit{FIN}^\alpha \longrightarrow \mathit{lec}\ \mathit{None}\ (\mathit{s2l}\ A)\ (\mathit{s2l}\ B) \\
\quad A \quad \sqsubseteq \quad B & = \dots & \mathit{lexord}\ (>)\ (\mathit{s2l}\ A)\ (\mathit{s2l}\ B)
\end{array} \tag{8}$$

3.3 Nesting and extensibility

Still, we cannot use RBTs for sets of sets of sets, as we have not yet instantiated *pi* for sets. To implement $\mathit{pi}^{\alpha\ set}$, we must also know α's cardinality – to that end, we use the overloaded constant $\mathit{card\text{-}UNIV}^\alpha$ from [12]. To see why cardinality matters, consider the sets $\emptyset$ and $\mathit{UNIV} = \overline{\emptyset}$. Now, $\mathit{pi}^{\alpha\ set}\ \lfloor\emptyset\rfloor\ \lfloor\overline{\emptyset}\rfloor$ holds iff there is a set A with $\emptyset \sqsubset A \sqsubset \mathit{UNIV}$ iff more than one value inhabits α. Yet, pi^α does not suffice to decide this: As we represent *UNIV* as $\overline{\emptyset}$, we do not get hold of any value of α, which we need for calling pi^α.

We now implement $\mathit{pi}^{\alpha\ set}$. The border cases are easy thanks to P1:

$$\mathit{pi}^{\alpha\ set}\ \mathit{None}\ \lfloor B \rfloor = (B \neq \emptyset) \qquad\qquad \mathit{pi}^{\alpha\ set}\ \lfloor A \rfloor\ \mathit{None} = (A \neq \mathit{UNIV})$$

However, we can compute $\mathit{pi}^{\alpha\ set}\ \lfloor A \rfloor\ \lfloor B \rfloor$ only if A and B are (co)finite, i.e., we need to pattern-match on the pseudo-constructors like in (8). Note that $\mathit{pi}^{\alpha\ set}\ \lfloor \mathit{Compl}\ A \rfloor\ \lfloor \mathit{Compl}\ B \rfloor = \mathit{pi}^{\alpha\ set}\ \lfloor B \rfloor\ \lfloor A \rfloor$ holds by P3. For the other cases, we define auxiliary functions *PI*, *PIc*, and *cPI*. For example,

$$\mathit{pi}^{\alpha\ set}\ \lfloor A \rfloor\ \lfloor \mathit{Compl}\ B \rfloor = \dots\ \mathit{PIc}\ \mathit{None}\ 0\ (\mathit{s2l}\ A)\ (\mathit{s2l}\ B) \tag{9}$$

where *PIc* satisfies (10) if α is finite, xs and ys are sorted and duplicate-free lists, and *set* $xs \cup$ *set* $ys \subseteq b{\uparrow}$; $||A||$ denotes the number of elements of the set A.

$$\mathit{PIc}\ b\ ||\mathit{UNIV} - b{\uparrow}||\ xs\ ys = (\exists A \subseteq b{\uparrow}.\ \mathit{set}\ xs \sqsubset_1 A \wedge A \sqsubset_1 \overline{\mathit{set}\ ys} \cap b{\uparrow}) \quad (10)$$

Like *cle*, the three functions traverse the list representations and call pi^α at most linearly many times. Their definitions are technical, but provide no new insights.

At last, we can order arbitrary nestings of sets and thus implement them as RBTs efficiently. Yet, the specification (6) for pi^α violates our rule of making sure that every type can be instantiated: They depend on the default linear order $<$, which does not work for α *set*. Therefore, we introduce another overloaded constant cpi^α – its specification is the same as (6) except that *corder* replaces $<$ and $\mathit{corder}^\alpha \neq \mathit{None}$ and FIN^α guard the equations. Hence, we have to implement cpi^α sensibly only for finite types α, for which we have provided an order, too. As most types are infinite, the restriction to finite types saves a lot of work.

4 Evaluation

To evaluate the efficiency and usability of our approach, we have performed two micro-benchmarks (§4.1 and §4.2) and integrated it with the Java interpreter (§4.3). All run-time tests ran on a Pentium DualCore E5300 2.6GHz with 2GB of RAM using Ubuntu GNU/Linux 9.10 and PolyML 5.4.1 or mlton 20100608; the figures are the average of four runs.

In preliminary tests, we noticed that Isabelle's default implementation based on lists sometimes outperformed LC with RBTs, even for large sets. We found that intermediate lists caused the slowdown. When we represent the sets as RBTs, *s2l* in code equations such as (8) and (9) first converts them into lists. While this simplifies the definitions and proofs, constructing the whole intermediate list is costly at run time - especially as the first few elements often suffice. Thus, we have manually eliminated such intermediate lists using the *destroy*/*unfoldr* pattern from shortcut fusion [18] before performing the benchmarks; we have proved the transformation correct in Isabelle.

4.1 Comparison with other approaches

The first micro-benchmark compares our approach with Isabelle's default implementation for sets, the ICF [9], and a conventional, RBT-based implementation. We start with the empty set, insert n numbers, then remove n numbers, then test n numbers for membership, and iterate over the set counting those elements less than n. All numbers are chosen randomly between 0 and $2n$ and implemented with ML's arbitrary precision integers. As discussed in [9], this benchmark measures the efficiency of the most common operations insertion, removal, membership and iteration, which all of the above implementations support.

Table 1 shows the run times under mlton for different n. As the first three rows all use RBTs, we can estimate the overhead that our approach (LC) and ICF add to a direct implementation with RBTs. LC's overhead is less than 1%,

Impl. / *n*	10k	20k	30k	40k	50k	100k	500k	1M	1.5M	2M	std. dev.
LC	.065	.138	.213	.295	.375	.825	5.17	10.8	17.0	23.3	< 2.0 %
ICF	.066	.139	.217	.300	.387	.839	5.17	11.8	17.4	23.9	< 1.4 %
RBT	.065	.135	.211	.292	.376	.818	5.59	10.8	17.0	23.3	< 2.3 %
default	1.50	5.72	3.08	22.8	36.2						<10 %

Table 1: mlton run times in seconds for the comparison benchmark; the last column bounds standard deviation (in percent of the run times) over all n.

the ICF's varies between 2% and 10%. The last row refers to Isabelle's (much slower) default setup with lists, which has quadratic complexity. Under PolyML, the RBT-based implementations take more than twice as long, and the average overheads are 2.5% for both ICF and LC. In conclusion, our approach is as efficient as the ICF and would be a good replacement for Isabelle's default setup.

4.2 Nested sets

This benchmark exercises the linear order on sets from §3. It starts with $\emptyset$ and inserts n sets to obtain a set $\mathcal{A}$ of sets. We generate each member set by inserting a random number of random numbers and randomly taking the complement; random numbers are chosen between 0 and m. Then, we check whether $\mathcal{A}$ contains another 100 sets generated the same way, and compute the size of the union of all sets in $\mathcal{A}$. We now use unsigned 32-bit words from Isabelle's word library for the numbers; so we exercise the *pi* implementation, too, as the word type is finite.

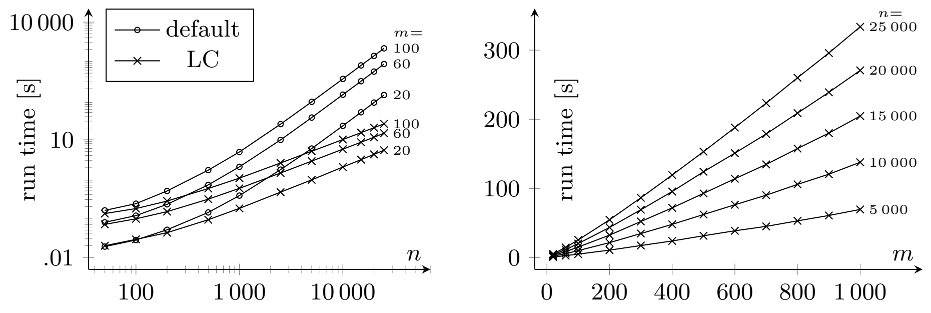

Fig. 1: mlton run times for the benchmark with nested sets

Figure 1 shows the mlton run times for Isabelle's default setup −∘− and ours −×− – like in §4.1, PolyML runs take twice as long. In the log-log plot on the left, m is fixed and n varies. From $n = 1000$ on, the plots are linear with slope 2 and 1, respectively. As the slope denotes the exponent of the polynomial complexity, this confirms that Isabelle's setup is quadratic and ours almost linear. While default is almost as fast as LC for small sets, LC is much faster for larger n and m. For $n = 25000$ and $m = 100$, e.g., it is 5.6 min vs. 35.6 min.

On the right, we now vary the size m of the inner sets for fixed numbers n of sets, but show only our approach LC. As the scales are linear, the plots fit a $m \log m$ curve. This shows that our linear order on α *set* is indeed efficient.

4.3 Case study: Java interpreter

In [15], we generated an interpreter for multithreaded Java from JinjaThreads, a large (86kLoC) Isabelle formalisation. Already for small programs, we achieved performance gains of 13% by pre-computing the lookup functions that extract information from the program declaration. We cached the information in (associative) lists using data refinement from α *set* and (α, β) *mapping*, Isabelle's type for finite executable maps. Type class restrictions disallowed more efficient implementations like RBTs, because the keys are (class) names, i.e., lists of chars, and ordering for lists has already been fixed to the (partial) prefix order.

Switching to our new approach was straightforward: we only had to import our new Isabelle files and instantiate the new type classes *ceq*, *corder*, and *set-impl* for the custom types as explained in §2. All existing definitions and proofs remained untouched. All in all, it took less than two hours. This shows that our approach is indeed easy to use.

To assess the impact on run-time, we took a Java program with 99 classes, converted it to JinjaThreads input syntax using Java2Jinja [13, §6.5], and ran the well-formedness checker (WF), the source code interpreter (SC), and the virtual machine (VM) on it. Table 2 shows the timings; the first row gives the timing for the original JinjaThreads versions without caching, the second with list-based caching. LC denotes our new approach: it gains 30% over the list-based implementation for WF, which heavily exercises the lookup functions and profits most from caching. As the interpreter calls the lookup functions less frequently, the gain is much smaller (1%). Surprisingly at first, LC ruins the VM performance – it is even slower than without caching. The bottleneck is the automatic selection of the set implementation. Internally, the VM uses sets also as a non-determinism monad the type of whose elements is built from 84 type constructors and 3 type variables. Hence, the execution of each bytecode instruction needs to query the available operations of all these constructors before it picks a set implementation. As a remedy, we disabled the automatic selection locally by replacing $\emptyset$ with *MSet* [] in the VM's code equations. This improves the run-time to .068 s and requires only three Isabelle declarations. To achieve the same performance as list-based tabulation (.053 s), we further replace the other operations on these sets with those that only have code equations for *MSet*. Hence, we save the dictionary constructions that emulate the type classes *ceq* and *corder* in ML.

	WF	SC	VM
w/o	2.51	5.81	.086
lists	.013	5.39	.053
LC	.009	5.35	.106 (.053)

Table 2: PolyML run times [s] for the Java interpreter

5 Conclusion and future work

We have proposed a light-weight approach to getting efficient code from Isabelle formalisations based on type classes and code generator refinement. It is flexible, extensible, and nestable. Our benchmarks and a case study show that it is indeed efficient, easy to use, and fits in the existing Isabelle libraries.

Four features of Isabelle's code generator have been crucial for this work:

Incremental declarations are the key to extensibility, as we can adjust the code generator setup right until code generation. This allows us, e.g., to bypass the impossibility results for single-parameter type classes [4,17].

Data refinement permits to represent values differently in logic and code. For configuration options for code generation, we suggest to take this to extremes (as shown in §2.3): Merge all cases in the logic! So, users can add further cases (data refinement) and change the configuration (program refinement).

Type classes enable overloading, our generated code uses them to query polymorphic type parameters. Type classes for code generation should be independent of those for logical concepts (e.g., *corder* vs. *linorder*, §2). For extensibility, they should be definitional such that every type can instantiate them.

Sequential pattern matching has helped to keep the effort linear in the number of implementations, see §2.4 and (8). This feature is hardly known; in Isabelle2013, such equations must be declared in the reversed order.

The next step is to cover more container types and implementations, e.g., bags and hash tables. Moreover, we want to integrate LC with Isabelle's packages such that they instantiate the type classes automatically. Also, we hope to equip Isabelle's code generator with an analysis that catches exceptions due to unsupported operations already at generation time. Future case studies will show how much effort LC saves in new developments.

Acknowledgements We thank D. Lohner and J. Breitner for valuable discussions and comments on earlier drafts and F. Haftmann and L. Bulwahn for helping with the code generator. This work has been partially funded by DFG grant Sn11/10-2.

References

1. Appel, A.W.: Efficient verified red-black trees. `http://www.cs.princeton.edu/~appel/papers/redblack.pdf` (2011)
2. Berghofer, S., Nipkow, T.: Random testing in Isabelle/HOL. In: SEFM'04, pp. 230–239. IEEE Computer Society (2004)
3. Berghofer, S., Reiter, M.: Formalizing the logic-automaton connection. In: TPHOLs'09. LNCS, vol. 5674, pp. 147–163. Springer (2009)
4. Chen, K., Hudak, P., Odersky, M.: Parametric type classes. In: LFP'92, pp. 170–181. ACM (1992)
5. Greve, D.A., Kaufmann, M., Manolios, P., Moore, J.S., Ray, S., Ruiz-Reina, J., Sumners, R., Vroon, D., Wilding, M.: Efficient execution in an automated reasoning environment. J. Funct. Program. 18(1), 15–46 (2008)

6. Haftmann, F., Krauss, A., Kunčar, O., Nipkow, T.: Data refinement in Isabelle/HOL. In: ITP'13. LNCS, vol. 7998, pp. 100–115. Springer (2013)
7. Haftmann, F., Nipkow, T.: Code generation via higher-order rewrite systems. In: FLOPS'10. LNCS, vol. 6009, pp. 103–117. Springer (2010)
8. Klein, G., Nipkow, T.: A machine-checked model for a Java-like language, virtual machine and compiler. ACM Trans. Progr. Lang. Sys. 28, 619–695 (2006)
9. Lammich, P., Lochbihler, A.: The Isabelle Collections Framework. In: ITP'10. LNCS, vol. 6172, pp. 339–354. Springer (2010)
10. Lammich, P., Tuerk, T.: Applying data refinement for monadic programs to Hopcroft's algorithm. In: ITP'12. LNCS, vol. 7406, pp. 166–182. Springer (2012)
11. Lescuyer, S.: Containers: a typeclass-based library of finite sets/maps. `http://coq.inria.fr/pylons/contribs/view/Containers/v8.3` (2011)
12. Lochbihler, A.: Formalising FinFuns – generating code for functions as data from Isabelle/HOL. In: TPHOLs'09. LNCS, vol. 5674, pp. 310–326. Springer (2009)
13. Lochbihler, A.: A Machine-Checked, Type-Safe Model of Java Concurrency : Language, Virtual Machine, Memory Model, and Verified Compiler. PhD thesis, Karlsruher Institut für Technologie, Fakultät für Informatik (2012)
14. Lochbihler, A.: Light-weight containers. Archive of Formal Proofs (2013) `http://afp.sf.net/entries/Containers.shtml`, Formal proof development.
15. Lochbihler, A., Bulwahn, L.: Animating the formalised semantics of a Java-like language. In: ITP '11. LNCS, vol. 6898, pp. 216–232. Springer (2011)
16. Marić, F.: Formal verification of a modern SAT solver by shallow embedding into Isabelle/HOL. Theor. Comput. Sci. 411(50), 4333–4356 (2010)
17. Peyton Jones, S.: Bulk types with class. In: Haskell Workshop 1997. (1997)
18. Svenningsson, J.: Shortcut fusion for accumulating parameters & zip-like functions. In: ICFP'02, pp. 124–132. ACM (2002)
19. Thiemann, R.: Generating linear orders for datatypes. Archive of Formal Proofs (2012) `http://afp.sf.net/entries/Datatype_Order_Generator.shtml`, Formal proof development.