A Universal Verification Framework for Curry

Fredrik Wieczerkowski

Master's Thesis
October 2025

Programming Languages and Compiler Construction
Department of Computer Science
Kiel University

Advised by
Prof. Dr. Michael Hanus
M.Sc. Kai-Oliver Prott

Abstract

The functional logic programming language Curry lets users program at a high level of abstraction, yet the type system is not strong enough to express e.g. the type of non-negative integers. While recent work has shown that external program verification tools on the intermediate language FlatCurry offer a flexible way of inferring and checking such constraints, developing such tools often requires implementing the same machinery around the traversal, transformation and caching of FlatCurry programs. In this thesis, we present a new framework for program verifications in the Curry programming language, based on FlatCurry and inspired by CASS. Our framework abstracts over the structure of verification tools like the Curry contract prover and provides a general foundation for structuring function-level verifications in the form of a fixed-point computation. We demonstrate the feasibility of this approach by porting the existing contract prover to our framework. We also implement a new tool based on the framework, namely a termination checker. While the idea is based on the termination analysis from CASS, we implement a more general termination criterion, the size-change principle, using the graph-based characterization by Lee et al. To infer size-change relations between functions and their calls, our tool uses both a subterm-based and an SMT solver-based strategy, which in conjunction allow it to prove termination in a wide variety of cases, including functions involving mutual recursion or integer arithmetic.

Acknowledgements

I am extremely grateful to my advisors, Prof. Dr. Michael Hanus and M.Sc. Kai Prott, for their invaluable advice, insights and feedback during this project. I also want to extend major thanks to my family and friends for patiently proofreading various parts of my thesis and for all the valuable suggestions.

Contents

List of Figures

Chapter 1

Introduction

Functional programming, with its emphasis on declarative, immutable and composable programs has proven to be a powerful programming paradigm that has gained much traction in recent years[1]. Functional programs are often shorter and easier to reason about than imperative programs, which is especially valuable in large or concurrent programs, as this greatly affects extensibility and maintainability. Pure and lazy functional languages like Haskell apply these principles rigorously by requiring all functions to be mathematically pure and side effects such as *Input/Output (I/O)* or mutable state to be encoded explicitly in the type signature.

Functional logic programming takes the ideas of functional programming further by adding features commonly known from logic programming, such as nondeterminism, free variables or unification. In this thesis we will focus on the language Curry, which can be viewed as an extension of Haskell that incorporates these logic programming features while retaining the feel of a functional language.

Both Haskell and Curry feature a strong type system that emphasizes algebraic data types, parametric polymorphism and type classes for encoding a wide variety of semantics at the type-level. In fact, the notion of type-driven programming has proven to be a useful model for making illegal states unrepresentable at compile-time and thereby eliminating a common source of run-time failures [Kin19]. A simple example of this is the notion of a non-empty list, which makes the empty list unrepresentable and could be defined as follows:

```
data NonEmpty a = a :| [a]
```

The general idea behind this is to refine a type to make a stronger statement about its value. This approach, however, has limitations, especially in a language like Curry. First, many refinements cannot be adequately expressed in Curry's type system. This includes all arithmetic constraints, such as an `Int` that must be non-negative. Secondly, the burden is on the programmer to explicitly specify these stronger types and to use them where applicable. For the former, refinement and dependent types, as implemented in proof assistants like Agda or Coq, solve this problem by generalizing the type system and converging the type and value language to allow defining such types, at the expense of complicating type inference and checking, along with being slower to compile.

While Haskell supports a subset of refinement types with the `LiquidHaskell` extension [VSJ+14; VSJ14], a different approach has emerged in the Curry ecosystem, namely the idea of external *program verification*: Instead of extending the type system directly, the inference and verification of program invariants is performed by a standalone tool that operates on the intermediate representation FlatCurry emitted by the Curry compiler frontend. This approach allows for greater flexibility as new verifications can be implemented independently of the compiler itself. One example of a verification is the so-called

[1]A good example for this is how heavily functional-influenced languages like Rust [KNK18] have made their way into the mainstream, with Rust gaining popularity for high-level systems programming to the point of being used in high-profile projects such as the Linux kernel [Tor22]. Rust shares many similarities with Haskell, OCaml and other members of the ML family of functional languages, including algebraic data types (enums), type classes (traits), type families (associated types) and more. Interestingly, there has even been research around implementing refinement types in Rust via a translation to Coq for program verification purposes, with focus on practical applications in the context of memory-unsafe code [GSJ+24].

contract prover, which verifies pre- and postconditions of functions using an external *Satisfiability Modulo Theories (SMT)* solver and serves a similar purpose as the aforementioned refinement types [Han17a]. In the context of Curry's nondeterminism, which includes the ability for functions to not return a value at all and thus "fail" the computation, external verification also provides a way of both checking and inferring conditions under which associated functions may not fail [Han18; Han24; Han25].

From an implementation perspective, implementing such verification tools involves a lot of common patterns, including resolving dependencies, simplifying the programs, performing a fixed-point iteration over all functions to compute associated verification results and potentially caching them to allow for better performance on incremental changes. While existing analysis frameworks for Curry exist, most notably the *Curry Analysis Server System (CASS)* [HS14], CASS is missing support for I/O and geared towards use as a server, hence it is not well-suited for larger verifications involving e.g. SMT-based computations. Still, its dependency-based function analysis provides a general model for abstracting over analyses which would prove useful in the context of verifications too.

1.1 Contributions

In this thesis we implement a new framework for program verifications in Curry that are structured in the form of function-level fixed-point computations. Our design draws inspiration from CASS, while abstracting over existing verification tools such as the contract prover or the non-failure verification tool. We demonstrate the feasibility of this approach by porting the contract prover to the framework. Additionally, we implement a new termination checker in terms of the framework to showcase a non-trivial example of a verification. As part of the termination checker we present an approach for using the size-change principle to prove termination for a wide range of different Curry functions, handling both mutual recursion and integer arithmetic through the integration of an SMT-based translation scheme.

1.2 Outline

We begin by providing some background about Curry and program verification in chapter 2. In chapter 3 we motivate and discuss the design of the verification framework at a high level, viewing the framework from a client perspective and treating the implementation as a black box. In chapter 4 we focus on the internal architecture of the framework, prominently including a detailed discussion of the verification runner and the caching mechanism. In chapter 5 we put the framework into practice by discussing how the existing verifications can be ported to the framework and introduce the new termination checker. Finally, we conclude the thesis with a summary and a brief outlook for future work in chapter 6.

Chapter 2

Preliminaries

We begin with an overview of the Curry programming language and its intermediate representation FlatCurry in section 2.1. Throughout the thesis we assume the reader to be familiar with functional programming, specifically pure functions and basic Haskell syntax, though we will discuss some higher-level concepts, including monads and monad transformers, as these will be essential to understanding our framework's design. Finally, we will provide some background on general concepts around program verification in the context of Curry, along with specific examples, in section 2.2.

2.1 Curry

Curry is a functional logic language that uses the syntax of Haskell and extends it with features known from logic programming, including nondeterminism, free variables and unification [Han13; Han16]. While Curry semantically shares many similarities with Haskell, including being lazy and pure, there are some important differences to be aware of.

2.1.1 Nondeterminism

Unlike Haskell, expressions in Curry may have zero or multiple values, a property known as *nondeterminism*. This is the result of replacing the fundamental expression evaluation strategy, which in Haskell is based on pattern matching and therefore causes every expression to deterministically evaluate to a single value[1], with *narrowing*, a rewriting strategy that replaces pattern matching with unification [Han13; Sla74]. The equations defining a Curry function are interpreted as a *Term Rewriting System (TRS)*, with their left-hand sides being successively unified with the expression to be evaluated, thereby yielding all possibly solutions in order[2]. A crucial difference to Haskell is that overlapping patterns in Curry functions introduce nondeterminism since *all* matching equations will be considered, rather than just the first one like in Haskell[3]. Consider the following example:

```
coin :: Bool
coin = True
coin = False
```

[1]The only exception being diverging, i.e. non-terminating or erroring, expressions.

[2]The idea is comparable to Prolog's SLD resolution, which also uses unification to match rules with a goal. There are some notable differences though: Instead of operating on predicates, Curry's narrowing operates on expressions like regular functional languages. Additionally, the form of narrowing used by Curry, namely *needed narrowing* [AEH00], is a lazier strategy than SLD resolution which always tries to reduce the goal eagerly.

[3]This also means that function rules may be reordered freely in Curry if the order of the solutions does not matter.

In Haskell, this example does not compile[4], whereas in Curry the expression `coin` would nondeterministically evaluate to both `True` and `False`. Since nondeterministic choice between two values is a common operation in Curry, the standard library's `Prelude` module defines the choice operator `(?)`:

```
(?) :: a -> a -> a
x ? _ = x
_ ? y = y
```

Using this operator, we can concisely define `coin` as `(True ? False)`. As mentioned above, nondeterminism also makes it possible for expressions to not evaluate to anything. This can happen when matching a value without a corresponding pattern[5] or when explicitly calling the primitive `failed` function from the `Prelude`.

It should be noted that from a syntax point-of-view, this model differs slightly from classical logic programming, as known from e.g. Prolog, which is structured around predicates instead of functions and expressions [WPP77]. Semantically, these models are equivalent, however, since multi-valued functions can be viewed as predicates by adding an additional parameter representing the output value, and vice-versa by expressing the predicate as a boolean function in terms of free variables. In fact, this equivalence is exploited by the two major Curry compilers, *Portland Aachen Kiel Curry System (PAKCS)* and *Kiel Curry System Version 2 (KiCS2)* [HAB+10; BHP+11], which compile Curry code to Prolog and Haskell, respectively, thus translating away either the logic-oriented or functional aspects of the language.

2.1.2 Referential Transparency

A central feature of functional languages that are both lazy and pure is *referential transparency*, namely the ability to freely replace expressions with their values without changing the parent expression's value or semantics. In a referentially transparent language like Haskell, for example, the expression `let x = f 42 in x + x` is fully equivalent to `f 42 + f 42`. It is easy to see why referential transparency requires purity: If `f` performs a side effect and the language permits this form of impurity, the second expression would run the effect twice. With laziness, the situation is a bit more subtle, but in order for referential transparency to work well, programs must not rely on their evaluation order and laziness effectively enforces this[6].

Curry largely inherits referential transparency from Haskell and this property does, in fact, hold for deterministic programs. There is, however, an important interaction with nondeterminism that needs to be considered, namely the distinction between *Call-Time Choice (CTC)* and *Run-Time Choice (RTC)*, which determines how an expression like `let x = 0 ? 1 in x + x` is to be interpreted. Under CTC semantics, the expression would be equivalent to `(0 + 0) ? (1 + 1)` and evaluate to `0` and `2`, whereas under RTC semantics, the expression would be interpreted as `(0 ? 1) + (0 ? 1)`, thus evaluating to `0`, `1`, `1` and `2`. Even though an interpretation using referential transparency would suggest RTC semantics, Curry uses CTC semantics to allow sharing the occurrences of the variable [BH05b; FKS11; Han16]. This also means that Curry is not referentially transparent in the strict sense. Still, as Fischer et al.

[4]In this specific case the two rules' left sides not just overlap, but are empty, which causes the Haskell compiler to interpret them as invalid multiple definitions of the function. In general, overlapping function rules, as long as they are non-empty, are allowed in Haskell and evaluate to the first match, as mentioned earlier.

[5]In Haskell, non-exhaustive pattern matching triggers a run-time error, whereas in Curry this will result in a failed evaluation, i.e. in having the expression nondeterministically evaluate to zero values. This subtle distinction can be useful when the user deliberately wants to discard a certain branch of the computation, even though e.g. other branches might yield a value.

[6]The relationship between purity and laziness is explored in the 2007 paper [HHPJ+07], which is also an interesting read on the history of Haskell. Peyton-Jones notes that even though purity does not require laziness, unlike the converse, the temptation to offer support for side effects is usually too great to resist in practice if the evaluation order can be relied on.

note, CTC provides a more intuitive programming model for lazy nondeterminism as variables can be interpreted as values rather than nondeterministic computations.

2.1.3 Normal Forms

When talking about evaluation semantics in a lazy language, it is also useful to understand the different kinds of normal forms, each of which standardize a certain form to which an expression has been reduced. The two most common normal forms are the *Weak Head Normal Form (WHNF)* and the *Normal Form (NF)*[7]. Intuitively, WHNF represents the evaluation to the outermost constructor or lambda expression, effectively a lazy evaluation step, while NF represents the full evaluation of an expression, essentially a strict evaluation. Formally, an expression is in WHNF when it either has the form `c x1 ... xn` for a constructor or primitive function `c` and expressions `x1, ..., xn` or the form `\p -> e` with `p` being a pattern and `e` an expression. An expression is in NF, on the other hand, when it cannot be reduced any further. It should be noted that every WHNF is also an NF, thus NF is a strictly stronger normal form [PJ87, pp. 197–199; Ses02]. To illustrate these normal forms using the example from [Jel18], consider the `replicate` function from the `Prelude`, which repeats an element `n` times and can be defined as follows:

```
replicate :: Int -> a -> [a]
replicate n l = if n <= 0 then [] else x : replicate (n - 1) x
```

Evaluating the expression `replicate 3 'a'` yields the following results:

```
WHNF:  'a' : replicate 2 'a'
  NF:  'a' : 'a' : 'a' : 'a' : []
 aka.  "aaaa"
```

Technically speaking, these normal forms are defined for function expressions too. For example, if we were to consider the WHNF and the NF for the partially evaluated call `replicate 3`, we would get the following terms:

```
WHNF:  \l -> if 3 <= 0 then [] else x : replicate (3 - 1) x
  NF:  \l -> x : x : x : []
```

While the WHNF only evaluates to the lambda abstraction, NF also evaluates the expression deeply under the lambda expression. It should be noted that this is more of theoretical interest and generally not an evaluation strategy employed in practice, as any actual evaluation to NF is generally done on expressions that evaluate to constructor terms, i.e. values. For our thesis it is, however, still of interest that function expressions have an NF as this will let us define termination in a more general fashion that lets us handle higher-order functions too. We will go more into detail on this in section 5.2.1.

It should also be noted that none of these normal forms are guaranteed to exist, especially when potentially infinite recursion is involved. Trivially, the value `let loop = loop in loop` has neither a WHNF nor NF, but more elaborate examples exist too. For example, the expression `[1..]` representing the infinite list starting at 1 has the WHNF `1 : [2..]`, but does not have an NF since it can always be reduced further.

Although the discussion above focuses on Curry, WHNF and NF likewise apply to FlatCurry. In fact, since FlatCurry has no lambdas, due to the lambda lifting performed by the Curry frontend that

[7]Technically, there also exists a *Head Normal Form (HNF)*, which is effectively a WHNF that includes evaluation under lambda abstractions, though still just to the outermost constructor. While reduction strategies involving HNF exist, they are less commonly used due to practical issues such as the presence of free variables [PJ87, pp. 199–200]. Although it may be interesting to investigate whether these arguments apply to Curry too, WHNF is still the more commonly used form for lazy evaluation.

transforms all lambdas to top-level functions, WHNF can be always defined as the evaluation to the outermost constructor.

2.1.4 Monads

As a pure language, Curry prohibits functions from performing arbitrary side effects, such as I/O or state mutations, by default. Though this may seem limiting at first, Curry, like Haskell, offers a solution that takes advantage of the strong type system to allow for explicit side effects without breaking purity, namely via *monads*. There are many ways to think about monads, including as flat-mappable wrappers around some type or as a generalization of list comprehensions [Wad90], and it can be useful to read different explanations to see which one works best for oneself. That said, in languages like Curry or Haskell it is often helpful to think of monads as a means of expressing and sequencing effectful computations, all without breaking purity or referential transparency[8]. To emphasize that they are not hidden as part of the implementation, we will refer to any monadic "side effects" solely as *effects* going forward. Since all Curry functions are mathematically pure, they have to declare any effects they wish to perform in their type signature.

To illustrate this, consider a function of the form `f :: Int -> Int`. In an impure language, `f` could do a lot of things that we would call side effects: It could read a file, mutate global state or invoke a random generator, possibly returning a different result each time it is invoked. In a pure language like Curry, the *only* thing `f` can do is to perform some pure operation on its input parameter, so that e.g. `f 42` would always evaluate to the same result[9]. This means that if we want `f` to be able to access files, for example, we would have to write it as `f :: Int -> IO Int`, i.e. wrap the result in the `IO` monad. Similarly, if we wish for the function to be able to access some global state `s`, we would have to write it as `f :: Int -> State s Int`.

At a technical level, a monad is a type constructor `m` that conforms to the following type class[10]:

```
class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
```

The `return` method wraps a pure value in a monadic context and the `(>>=)` method, commonly known as *bind* or *flatMap* in other languages, maps a function with a monadic result over a monadic value. Common examples of monads are `Identity`, `IO`, `Maybe`, `[]`, `Except e`, `Reader r`, `Writer w` and `State s`. Notably, while each of these types are wrappers around some (possibly none or multiple) values of type `a`, they can be also be viewed as a computation returning an `a`, capturing a distinct effect:

- ▻ An `Identity a` is isomorphic[11] to an `a` and encapsulates a pure computation without any effects.
- ▻ An `IO a` is a value of type `a` that may perform I/O in its implementation.

[8]Note that the caveats around nondeterminism as outlined in section 2.1.2 still apply. It should further be noted that nondeterminism is sometimes considered a side effect in other languages. We will, however, exclude Curry's ambient, CTC-based nondeterminism from our definition of side effects for the purpose of defining monads. On an even more technical note, the list monad `[a]` essentially implements a separate form of nondeterminism at the language level in both Haskell and Curry, so in that sense nondeterminism, as a general concept, *can* sometimes be considered a side effect in the monadic sense. The details are out-of-scope and not relevant for the purposes of this thesis, however.

[9]There is an escape hatch to this, namely `unsafePerformIO` which allows performing side effects by letting the programmer unsafely evaluate an action in the `IO` monad, thus making it the programmer's responsibility to uphold purity. For our intents and purposes, we consider this to be an implementation detail, however, and not relevant to understanding monads in general.

[10]We omit the `Functor` and `Applicative` requirements here for brevity. The instances for both of these type classes can be derived directly from the `Monad` instance by setting `fmap = liftM`, `pure = return` and `(<*>) = ap`.

[11]For our purposes, we consider types to be *isomorphic* when they structurally represent the same data type. For example, the type `T a` defined by `data T a = X a | Y` would be isomorphic to `Maybe a`.

- ▻ A `Maybe a` is an optional value or in other words, a *possibly failing* computation.
- ▻ A `[a]` is a list of values `a` or, from an effects perspective, a *nondeterministic* computation[12].
- ▻ An `Except e a` is isomorphic to an `Either e a` and represents a computation that may *throw* an error `e`.
- ▻ A `Reader r a` is isomorphic to an `r -> a`, i.e. a computation that *reads* an environment `r`.
- ▻ A `Writer w a` is isomorphic to an `(a, w)`, i.e. a computation that *writes* an additional value `w`[13].
- ▻ A `State s a` is isomorphic to an `s -> (a, s)`, i.e. a computation that both *reads* and *writes* a state `s`.

While these monads provide a nice abstraction for performing a number of different effects, they leave an important question unanswered, namely the one of how we can use multiple effects at once. Going back to our example `f`, these monads alone provide no way for the function to, for example, both perform I/O and throw an error `e`. The solution to this problem are *monad transformers*, a generalization of monads that enables composition of different effects [LHJ95]. The key idea is to parameterize monads over some inner monad, so they can be "stacked together". Every monad listed earlier, except[14] for `IO` and `[a]`, naturally generalize to a transformer version, which, by convention, is suffixed with the letter `T`:

- ▻ `Identity a` becomes `IdentityT m a`
- ▻ `Maybe a` becomes `MaybeT m a`
- ▻ `Except e a` becomes `ExceptT e m a`
- ▻ and so on...

For every monad `m`, applying a transformer `t` to `m` will yield a new monad `t m` adding the corresponding effect. Colloquially, we also say that the transformer `t` has been "stacked" on top of `m`. For example, applying `MaybeT` to `IO` will yield a new monad `MaybeT IO` that encapsulates both failure and I/O. By stacking transformers like this, we can mix-and-match effects to create a monad that supports everything we need. In practice, a common convention is to abstract complex transformer stacks with a type synonym, conventionally suffixed with the letter `M` to clarify that it is a monad:

```haskell
type MyCustomM = ExceptT String (StateT [Int] IO)
```

Then we could write `f :: Int -> MyCustomM Int` to "equip" our function from `Int` to `Int` with the ability to throw errors of type `String`, access mutable state of type `[Int]` and perform I/O.

From an implementation perspective, monad transformers are often defined as `newtype`s that insert the `m` parameter at an appropriate position into the type signature of the original monad. For `Maybe` or `Except`, the corresponding transformers effectively just wrap the monad in `m`:

```haskell
newtype MaybeT    m a = MaybeT  { runMaybeT  :: m (Maybe a) }
newtype ExceptT e m a = ExceptT { runExceptT :: m (Either e a) }
```

[12]Again, the nondeterminism of the list monad should not be confused with the ambient nondeterminism of Curry that we introduced earlier in section 2.1.1.

[13]Technically, `w` also has to conform to the `Monoid` type class to make `Writer w a` a monad. In practice this means that values of type `w` must be composable in some associative way, for example by being a list type.

[14]The reason for why these monads have no transformer equivalent is because we wish every transformed monad to still be a monad, which does not necessarily hold in these cases. For `IO` this is rarely the case, see [McC12], whereas with `ListT` the situation is a bit more nuanced and there are situations in which a `ListT` can be useful, even if it technically does not satisfy the monad laws. See [GGB+] for further discussion.

Transformers like `ReaderT`, `WriterT`[15] or `StateT`, on the other hand, are defined directly, given that only the output of a monadic function has to be wrapped in the parameter monad `m`:

```
newtype ReaderT r m a = ReaderT { runReaderT :: r -> m a }
newtype WriterT w m a = WriterT { runWriterT ::      m (a, w) }
newtype StateT  s m a = StateT  { runStateT  :: s -> m (a, s) }
```

It should be noted that a transformer can always be turned back into a monad by applying it to `Identity`. For example, `MaybeT Identity` is isomorphic to `Maybe` when considering its monad semantics. This is especially relevant for the aforementioned transformers, since a `(ReaderT r m a)` aka. `r -> m a` is semantically different from an `m (Reader r a)`, aka. `m (r -> a)`. In these cases, it is convenient to derive the monad directly from the transformer, in this case by defining `Reader` as `ReaderT Identity`.

There is, however, still a missing piece in the puzzle: If we consider that I/O actions return values wrapped in `IO`, it might not be obvious how we would bind these in the `MyCustomM` monad. This is where the `MonadTrans` class comes in, a type class that all monad transformers implement:

```
class MonadTrans t where
  lift :: Monad m => m a -> t m a
```

The class provides a `lift` function that takes a value from the wrapped monad and turns it into a value of the transformed monad, i.e. "lifts" it up one level in the transformer stack. In our example, we have two layers of transformers, so applying `(lift . lift)` would turn an `IO` value into a `MyCustomM`[16].

There are a few important considerations to keep in mind when working with monad transformers: First, the `IO` monad always has to be at the bottom of the transformer stack as there is no `IOT` transformer per [McC12]. Secondly, the transformers do not always commute with each other, i.e. the order of the effects matters: The monad `ExceptT String (Writer [String])`, for example, is semantically different from `WriterT [String] (Except String)`, as the former will return the written `[String]` even if an error is thrown whereas the latter will only return the `[String]` if no errors are thrown[17].

2.1.5 FlatCurry

Curry features a wide ecosystem of compiler tools, including the PAKCS and KiCS2 compilers and a number of optimizers and analyzers. To abstract away the nuances of source-level representations of Curry, these tools share a compiler frontend that emits an intermediate language named *FlatCurry* [AHH+05; Han13; Han17c; Han17b], abbreviated with its file extension .fcy for untyped or .afcy for type-annotated FlatCurry. Figure 2.1 contains an overview of this compilation pipeline.

[15]Strictly speaking, `WriterT` could also be defined as `m (Writer w a)`. It is commonly defined via its transformer too, however.

[16]Things become slightly more complicated when attempting to "lift" a monad with a different ordering of effects into another, e.g. an `Except String` into our `MyCustomM`. While it is usually possible, especially if the effects commute, it requires wrapping and unwrapping the transformer implementations explicitly. The general solution for this problem are monad classes, which provide a generic interface for each monad, e.g. a `MonadExcept` class providing a generic `throwError` method or a `MonadState` class providing `get` and `put`. This way, each class captures an effect and functions can be parameterized over these classes, allowing any monad to be passed in, as long as it provides the corresponding effects. These require *Multi-Parameter Type Classes (MPTCs)*, which are a relatively recent addition to Curry [Krü21] and would have to be ported from Haskell, which has not been done yet in the standard monad transformer package for Curry.

[17]Anecdotally, this distinction actually came up in practice in the context of the Curry compiler frontend: If we consider the `String` to be an error message and the `[String]` to be a list of warning messages, then the former monad would allow emitting both warnings and errors, whereas the second monad would discard all warnings in the event that an error occurs. To make sure the compiler and other tools built on top of the frontend, like the Curry language server, emit as much information as possible, the monad in the frontend was at one point changed to the former definition.

[18]It should be noted that the syntax presented here is chosen for easy readability. Internally, FlatCurry modules use a more compact, machine-readable format that is conceptually of the same structure.

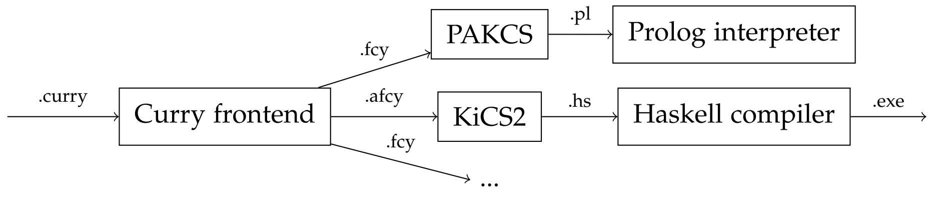


Figure 2.1. Compilation pipeline for Curry programs

$$
\begin{array}{rll}
P & ::= D_1 \ldots D_m & \text{(program)} \\
D & ::= f(x_1, \ldots, x_n) = e & \text{(function definition)} \\
e & ::= x & \text{(variable)} \\
 & \mid\ c(e_1, \ldots, e_n) & \text{(constructor call)} \\
 & \mid\ f(e_1, \ldots, e_n) & \text{(function call)} \\
 & \mid\ \textbf{case}\ e\ \textbf{of}\ \{p_1 \to e_1; \ldots; p_n \to e_n\} & \text{(case expression)} \\
 & \mid\ e_1\ \textbf{or}\ e_2 & \text{(disjunction)} \\
 & \mid\ \textbf{let}\ \{x_1 = e_1, \ldots, x_n = e_n\}\ \textbf{in}\ e & \text{(let binding)} \\
p & ::= c(x_1, \ldots, x_n) & \text{(pattern)}
\end{array}
$$

Figure 2.2. Syntax for the intermediate language FlatCurry[18]

FlatCurry can be considered a stripped-down version of Curry that, while still being high-level, already assumes a number of desugarings to have been applied, thus simplifying analysis and code generation. Specifically, FlatCurry only allows matching one level of constructors per case expression, requires recursion to be performed at the top-level[19], expresses functions with a single rule per definition and represents nondeterministic choices using *Or*-nodes, also known as *disjunctions*. An overview of FlatCurry's syntax, presented informally in a *Backus-Naur-Form (BNF)*-like notation, can be found in figure 2.2. Like [Han18] we omit the distinction between rigid and flexible case expressions here, which is not relevant for our purposes.

The type definitions for representing FlatCurry syntax trees in Curry itself are included in appendix A. A program P, for example, is represented by a `Prog` and a function declaration D by a `FuncDecl`, other syntax elements have similar analogues. The referenced definitions also include a number of helper types, like `QName` for representing module-qualified identifiers. While these type names can be considered implementation details, we will use this Curry type notation more extensively in later chapters, especially when discussing specific type signatures.

To understand the abstraction level of FlatCurry as an intermediate language, it is illustrative to take a concrete example, e.g. a Curry function expressing the factorial on non-negative integers:

```
fac :: Int -> Int
fac n | n == 0 = 1
      | n >  0 = n * fac (n - 1)
```

The Curry frontend translates this to the following FlatCurry function[20]:

[19]As opposed to Curry (and Haskell), which allow local bindings, e.g. in `let` expressions to be recursive.

[20]Note that the operators have been written in source-like infix notation for better readability. The actual FlatCurry output would e.g. include function calls to `_impl#+#Prelude.Num#Prelude.Int#` rather than `(+)`.

$$\text{(Val)} \qquad \Gamma : v \Downarrow \Gamma : v \qquad \text{where } v \text{ is constructor-rooted}$$

$$\text{(VarExp)} \qquad \frac{\Gamma : e \Downarrow \Delta : v}{\Gamma[x \mapsto e] : v \Downarrow \Delta[x \mapsto v] : v}$$

$$\text{(Fun)} \qquad \frac{\Gamma : \rho(e) \Downarrow \Delta : v}{\Gamma : f(\overline{x_n}) \Downarrow \Delta : v} \qquad \text{where } f(\overline{y_n}) = e \in P \text{ and } \rho = \{\overline{y_n \mapsto x_n}\}$$

$$\text{(Let)} \qquad \frac{\Gamma[\overline{y_k \mapsto \rho(e_k)}] : \rho(e) \Downarrow \Delta : v}{\Gamma : \textbf{let } \{\overline{x_k = e_k}\} \textbf{ in } e \Downarrow \Delta : v} \qquad \text{where } \rho = \{\overline{x_k \mapsto y_k}\} \text{ and } \overline{y_k} \text{ are fresh variables}$$

$$\text{(Or)} \qquad \frac{\Gamma : e_i \Downarrow \Delta : v}{\Gamma : e_1 \textbf{ or } e_2 \Downarrow \Delta : v} \qquad \text{where } i \in \{1, 2\}$$

$$\text{(Select)} \qquad \frac{\Gamma : x \Downarrow \Delta : c(\overline{y_n}) \quad \Delta : \rho(e_i) \Downarrow \Theta : v}{\Gamma : \textbf{case } x \textbf{ of } \{\overline{p_k \to e_k}\} \Downarrow \Theta : v} \qquad \text{where } p_i = c(\overline{x_n}) \text{ and } \rho = \{\overline{x_n \mapsto y_n}\}$$

Figure 2.3. Natural semantics of (normalized) FlatCurry

```
fac(v1) = case (v1 == 0) of
            True -> 1
            False -> case (v1 > 0) of
              True -> v1 * fac(v1 - 1)
              False -> failed
```

As we can see, the function guards have been transformed into case expressions that each match a single level of constructors. Notably, there are no missing branches here either: While the Curry definition did not handle the `n < 0` case explicitly, FlatCurry requires handling every constructor and therefore includes explicit `failed` calls for unmatched patterns.

FlatCurry is also used to specify Curry's operational semantics, defining e.g. how expressions are evaluated. For this we consider the notion of *normalized* FlatCurry programs, in which the *discriminating expression*, i.e. the expression being pattern-matched on, in every `case` is a variable. We can consider FlatCurry programs to be normalized without loss of generality, since any discriminating expression can be lifted into a surrounding `let` declaration.

FlatCurry's semantics can be considered an extension of Launchbury's natural semantics for lazy functional languages, adding support for nondeterminism via an additional (Or) rule [Lau93; AHH+05]. The specification for FlatCurry, as shown in figure 2.3, is given in mathematical inference rule notation and uses judgements of the form $\Gamma : e \Downarrow \Delta : v$ to relate an expression e in a *heap*[21] Γ to a value v, with Δ being the heap after evaluation. Formally, a heap Γ is a partial mapping from bound variables to expressions and we use the notation $\Gamma[v \mapsto e]$ to denote an updated heap where $v' \mapsto \Gamma(v')$ for all $v' \neq v$ and $v \mapsto e$. Due to nondeterminism, an expression may evaluate to multiple values, so

[21] In literature on type systems and type inference, it is common to talk about *contexts* or *environments* instead of heaps. We will, however, use the term *heap* here to be consistent with other Curry and FlatCurry literature [AHH+05; Han17a].

multiple judgements may hold on an expression e in a heap Γ. The notation $\overline{x_n}$ is used to concisely denote a finite sequence $x_1, ..., x_n$. Intuitively, the rules from figure 2.3 can be interpreted as follows[22]:

▷ The (Val) rule describes that constructor-rooted expressions, i.e. expressions in WHNF, evaluate to themselves, since they already can be considered values.

▷ The (VarExp) rule describes that a variable v that is already bound in Γ to an expression e should be evaluated separately to v and returned, along with updating the mapping in the resulting heap to v.

▷ The (Fun) rule specifies that to evaluate a function call of the form[23] $f(\overline{x_n})$, we first need to find a matching function declaration of the form $f(\overline{y_n}) = e$ in the program P, substitute the passed arguments $\overline{x_n}$ for the parameters $\overline{y_n}$ occurring in the function body e and then evaluate this expression, returning the updated heap Δ with the value v.

▷ The (Let) rule effectively works the other way around: To evaluate an expression of the form **let** $\{\overline{x_k = e_k}\}$ **in** e, we first create fresh variables $\overline{y_k}$, bind them in the heap to $\overline{x_k}$, replace occurrences of each x_k with y_k in the expression and then evaluate it. The updated heap Δ, along with the resulting value v, is returned again.

▷ As mentioned earlier, the (Or) rule is specific to Curry and specifies that for an expression e_1 **or** e_2, i.e. nondeterministic choice between e_1 and e_2 we should simply evaluate and return each[24].

▷ Finally, the (Select) rule describes how a case expression of the form **case** x **of** $\{\overline{p_k \to e_k}\}$ is to be evaluated in a heap Γ: First, the discriminating variable x is evaluated to WHNF in Γ, yielding a value $c(\overline{y_n})$, where c is the outermost constructor and an updated heap Δ. Then, the matching case arm for the constructor c is looked up, with the corresponding pattern $p_i = \overline{x_n}$, whose variables $\overline{x_n}$ are finally substituted into the body of the matching case arm e_i and evaluated in the updated heap Δ. This yields a final heap Θ and a value v, both of which are returned.

2.1.6 Curry Package Manager

Like Haskell, Curry offers a module system for splitting up source code into separate files, along with an import and export system for functions and types. As projects grow, dependency management quickly becomes cumbersome and, like in other languages, the need for a higher-level abstraction arises. The *Curry Package Manager (CPM)* solves this by providing a standardized way of packaging Curry projects [Obe16a]: Projects can declare a `package.json` at their root that includes basic metadata, such as the name or version of the project, a list of other packages they depend on and any executables the project provides. Using the `cypm` command-line tool, the project can then be built, tested and installed in a standardized way. Curry packages are published to a central repository at `https://cpm.curry-lang.org`, which CPM uses to resolve, install or upgrade any packages that are not already installed locally.

[22]It is often helpful to read formal inference rules as a procedure or algorithm, starting with "inputs" on the left-hand side of the bottom judgement, then "executing" the preconditions on the top and finally "returning" the right-hand side of the bottom judgement. This interpretation, in fact, is very similar to how one would implement functional programs in a logic programming language like Prolog.

[23]Note that "$f(\overline{y_n}) = e$" is to be understood as a purely formal expression and not as an equation or statement. In other words, the "=" symbol is used solely to mirror the FlatCurry syntax for a function declaration, not to state an equivalence.

[24]To keep thinking about inference rules algorithmically, this is the point where we would need to relax our intuition to have them represent relations instead of functions. Incidentally, this is precisely the difference between functions in a functional logic language like Curry and a traditional functional language.

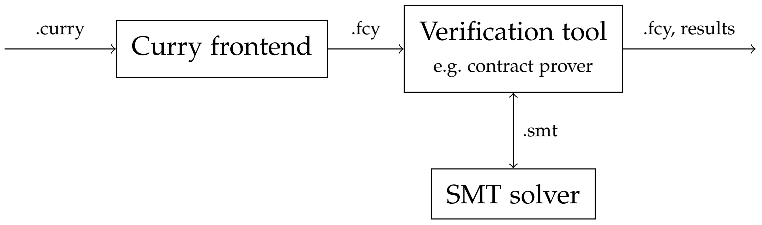


Figure 2.4. Abstract verification pipeline for Curry programs

Much of the infrastructure around invoking the frontend and dealing with Curry and FlatCurry syntax has been released in the form of Curry packages. This includes the `frontend-exec` package for compiling Curry source code to FlatCurry using the frontend, the `flatcurry`, `flatcurry-annotated` and `flatcurry-type-annotated` packages for reading, transforming and writing the different flavors of FlatCurry and the `flatcurry-smt` package, which includes some helpers for mapping Curry operations to SMT names. The gist here is that much of the Curry compiler ecosystem has been built around small, composable Curry packages, which we will also make heavy use of.

2.2 Program Verification

The factorial function introduced in section 2.1.5 provides an example and motivates the notion of program verification: Curry's type system is not strong enough to restrict a type e.g. to the set of non-negative integers[25], thus it is up to the programmer to make sure that every call to `fac` satisfies this precondition. In practice, this is prone to mistakes, as it is easy to accidentally pass an invalid value, even more so when the value is computed at run time.

Program verification provides a solution to this problem, by statically verifying these invariants and falling back to run-time checks where they cannot be proven. As evidenced by recent work by Hanus, implementing such verifications in the form of external tools has proven to be a flexible way of integrating them into the existing ecosystem of analysis and optimization tools for Curry [Han18; Han17a; Han24; Han25]. Like PAKCS and KiCS2, verification tools operate on FlatCurry modules emitted by the frontend. Unlike these compilers, the verification tools primarily output domain-specific verification results instead of generating code, they may, however, as a side effect, also modify the FlatCurry module to insert dynamic checks. Figure 2.4 provides an overview of the compilation pipeline for verification tools.

2.2.1 SMT Solving

Before discussing specific examples of program verifications, we will briefly review a key technique used by verification tools, namely SMT solving, as hinted at in figure 2.4. At its core, this refers to solving the SMT problem, a decision problem that generalizes the classic *Boolean Satisfiability Problem*

[25]While there are workarounds to statically enforce this in this case, such as using Peano numbers, which can be defined as purely algebraic data types, these are often much less efficient than using the native integer representation. Dependently typed programming languages, as mentioned in the introduction, provide an alternative solution to this, but we will focus on the verification-based approach in this thesis.

(SAT)[26] for some particular instance. Both SAT and SMT ask whether there is a variable mapping that makes a given Boolean-valued term true, i.e. *satisfies* it, SAT, however, only permits Boolean-valued variables whereas SMT also permits numbers, lists and other forms of structured data. This is particularly useful in the context of program verification as predicates on Curry expressions can in many cases be directly translated to SMT, therefore letting verifications take advantage of the robust algorithms for solving SMT for a wide variety of formulas in popular solvers like Z3 [DMB08].

The SMT-LIB Language

For the practical problem of communicating the input to an SMT solver, many solvers have adopted the SMT-LIB language [BST10], a domain-specific dialect of Lisp that defines a standardized syntax for declaring free variables, expressing assertions and checking satisfiability. While the full syntax can be found in [BST10], we will provide a brief overview of the most important constructs.

The central commands of SMT-LIB are `(declare-const <name> <sort>)`[27], which declares a variable of a specific *sort*[28] such as `Int`, `(assert <term>)`, which adds the given Boolean-valued term to the internal set of assertions and `(check-sat)`, which checks whether all assertions are satisfiable together, i.e. whether there is an assignment of the variables to make all assertions hold simultaneously. To illustrate this, consider the following example:

```
(declare-const x Int)
(assert (= (+ x 2) 8))
(check-sat)
```

This SMT-LIB program asserts the formula $x + 2 = 8$ with x being a free variable of sort Int and checks satisfiability. Invoking Z3 on this program yields the output `sat`, which tells us that this program is indeed satisfiable, and if we append the `(get-model)` command, Z3 will also output an assignment that satisfies the formula:

```
(
  (define-fun x () Int
    6)
)
```

As expected, $x = 6$ is the only solution in this case. For a more complex example, consider the formula of the golden ratio, namely $\frac{a+b}{a} = \frac{a}{b}$ where $a > b > 0$. Translated to SMT-LIB, we get the following program, which already looks relatively difficult to solve, given the nonlinear equation and real numbers involved:

```
(declare-const a Real)
(declare-const b Real)
(assert (> a b))
(assert (> b 0))
```

[26]SAT is particularly notable for to its central importance in complexity theory due to being the first NP-complete problem per the Cook-Levin theorem [Coo71; Kar72]. Given that therefore all NP problems, i.e. problems whose solution can be verified in polynomial time, can be reduced to SAT, SAT (and by extension SMT) solvers in particular have numerous applications in the field of optimization problems. We will, however, not go too far into detail on these and focus on our particular application in the context of program verification.

[27]The SMT-LIB language is actually slightly more general and permits declaring free functions via `(declare-fun)`, `(declare-const)` is effectively just the special case where the arity is zero. For our purposes, however, variables will be sufficient.

[28]In the context of SMT, *sort* is a synonym for *type*.

```
(assert (= (/ (+ a b) a) (/ a b)))
(check-sat)
(get-model)
```

However, Z3 is indeed powerful enough to solve this, as it outputs `sat` with the following model:

```
(
  (define-fun b () Real
    1.0)
  (define-fun a () Real
    (root-obj (+ (^ x 2) (* (- 1) x) (- 1)) 2))
  ...
)
```

The positive root of this quadratic polynomial, $x^2 - x - 1$, precisely corresponds to the golden ratio.

A convenient feature of SMT-LIB is that is supports scoping via the `(push)` and `(pop)` commands. The central idea is that the SMT solver internally not just stores an assertion set, but in fact a stack of assertion sets that can be pushed to and popped from to add/remove assertions during the script. For example, if we wish to verify $x = 42$ and both $x > 0$ and $x < 0$ separately, we can do so, but leveraging scopes:

```
(declare-const x Int)
(assert (= x 42))
(push)
  (assert (> x 0))
  (check-sat)
(pop)
(push)
  (assert (< x 0))
  (check-sat)
(pop)
```

During the first `(check-sat)`, the solver's stack contains the assertions $x = 42$ and $x > 0$, which are true, whereas during the second `(check-sat)`, the solver's stack contains $x = 42$ and $x < 0$, which are false, thus the output is, perhaps not unexpectedly `sat` and `unsat`.

2.2.2 Contract Checking

As mentioned earlier, a central application of SMT solving and one of the two concrete example verifications that we will take a closer look at in this thesis is the problem of checking pre- and postconditions of functions, i.e. invariants in the form of Boolean predicates over the function's arguments and, in the case of postconditions, also their return value [AH12].

The *contract prover* introduced in [Han17a] implements this verification, along with providing a corresponding tool. More precisely, it operates on a generalized notion of pre- and postconditions, namely *contracts*, which is a general term for all invariants upheld by a function. In Curry, the standard convention for contracts is to name them after their associated function. Specifically this amounts to the function's name suffixed with `'pre` for a precondition, `'post` for a postcondition and `'spec` for a specification[29]. Taking the factorial function introduced earlier as an example, we can formalize the precondition that the factorial is only defined for non-negative integers as follows:

[29]A specification is another kind of contract that expresses a certain input/output behavior of a function, i.e. effectively implements a restriction of the function. For more detailed discussion, see [AH12].

$$\text{(FunCheck)} \quad \frac{\Gamma : f\texttt{'pre}(\overline{x_n}) \Downarrow \Gamma' : \textbf{True} \quad \Gamma' : \rho(e) \Downarrow \Delta' : v \quad \Delta' : f\texttt{'post}(\overline{x_n}, v) \Downarrow \Delta : \textbf{True}}{\Gamma : f(\overline{x_n}) \Downarrow \Delta : v}$$

$$\text{where } f(\overline{y_n}) = e \in P \text{ and } \rho = \{\overline{y_n \mapsto x_n}\}$$

Figure 2.5. Extension of the FlatCurry's natural semantics to verify pre- and postconditions

```
fac'pre :: Int -> Bool
fac'pre n = n >= 0
```

Similarly, we can state in a postcondition that the output of the factorial is always $\geqslant 1$:

```
fac'post :: Int -> Int -> Bool
fac'post _ n = n >= 1
```

Formally, ensuring that the pre- and postconditions of a function hold amounts to extending the semantics of FlatCurry by replacing the (Fun) rule with a (FunCheck), as shown in figure 2.5 [Han17a]. The idea is to require the precondition f`'pre` to hold before evaluation and to require the postcondition f`'post` to hold after evaluation of f. It should be noted that while the precondition only takes the inputs $\overline{x_n}$ of the function f, the postcondition takes both the inputs and the output v. In the factorial example given earlier, the input is a single `Int` and the output is another `Int`.

A key difference between pre- and postconditions is that while preconditions verify the *usage* of a function, postconditions verify the *implementation*. Thus, a postcondition only has to be proven once, namely for the function's implementation, whereas a precondition has to be checked at every call site of the function. This principle guides the internal pipeline of the contract prover, which is to first check all postconditions and then the preconditions on every call, inserting a run-time check whenever the contract cannot be verified statically.

To perform the static verification, the contract prover traverses the program[30], collecting statically known invariants called *assertions* throughout the program[31] and checks whether these assertions cover the pre- and postconditions by using an SMT solver like Z3. If a postcondition or a precondition at a specific call site cannot be proven statically, a dynamic check is inserted to enforce it at run time. The use of an external solver has the advantage of leveraging existing methods for verifying not just algebraic, but also integer constraints, as needed e.g. for our factorial function. Simultaneously, it also forms a central requirement for our verification framework, namely the ability for verifications to perform I/O internally.

2.2.3 Non-Failure Verification

Aside from contract checking, there is another example of a verification that we will discuss. As noted in section 2.1.1, Curry's nondeterminism provides a first-class model for partial functions by allowing computations to fail throwing a run-time error. This is useful for logic-oriented computations, but also introduces potential for bugs. For example, consider the `head` function from the `Prelude`, which returns the first element of a list iff it is non-empty:

[30]Specifically, the right-hand side, i.e. the body expression, of each function's rule is traversed.

[31]The precise assertion-collecting semantics can be found in [Han17a]. We consider them to be an implementation detail, since the framework will operate at a higher level, specifically at the function-level, not at the expression-level.

```
head :: [a] -> a
head (x:_) = x
```

It is not safe to call this function on any list, since an expression like `head []` will compile without issues yet fail at run time. If the programmer can guarantee that `head` will always be called on a non-empty list, however, it can still be perfectly failsafe, even though the compiler cannot prove this[32]. The verifier introduced in [Han18] addresses this by introducing the concept of *non-failure conditions*, along with strategies for proving them. Like preconditions, non-failure conditions are boolean predicates on an associated function's parameters and express a sufficient condition under which the function is guaranteed not to fail. While related, non-failure conditions are semantically weaker than preconditions, since failing may be intentional[33]. Going back to the `head` example discussed earlier, the associated non-failure condition could be expressed as follows:

```
head'nonfail :: [a] -> Bool
head'nonfail xs = not (null xs)
```

Similar to the contract prover, the non-failure verifier in its basic form traverses the program, collecting assertions, then uses an SMT solver to prove these assertions. If the proof succeeds, the program is guaranteed not to fail at run time. While the precise assertion-collecting semantics are formalized in [Han18], the central idea is to prove that for every function call $f(\overline{x_n})$, either f`'nonfail`$(\overline{x_n})$ holds or that the call fails.

A more advanced form of non-failure verification is discussed in [Han24], which introduce the notion of *abstract types*, specifically *call types* and *I/O types*, to model the "shape" of the accepted and returned data terms. A depth-1 call type, for example, precisely represents the set of constructors accepted and returned by a function[34]. This technique makes it possible to infer and check non-failure conditions automatically for many Curry functions. [Han25] further generalizes the inference of non-failure conditions to functions involving integer arithmetic. The key takeaway for our verification framework is that inferring a call type or non-failure condition for a function effectively involves *computing* a value to be associated with the function.

[32]Note that there are other strategies for making the `head` function failsafe too. We could turn it into a total function by returning a `Maybe a` or by introducing a new data type for non-empty lists. Like with the factorial example earlier, introducing additional abstractions often incurs a performance or complexity cost in practice. Our verification-based approach both more lightweight and more general, since it applies to all algebraic data types.

[33]As long as any potentially failing computations are encapsulated, a program can still be verified as non-failing at the top-level, for example, therefore the non-fail condition cannot be understood as a precondition, which has to hold under all circumstances. Contracts can, however, aid in the verification of non-failure conditions [Han17a].

[34]Technically a depth-1 call type also includes the type $\top$ that includes all possible data terms. The intuition, however, is that a depth-1 type represents a single level of constructors and every additional level of depth adds another level of constructors.

Chapter 3

Design

In this chapter we will discuss the key ideas behind our framework at a high level. This includes understanding the challenges in writing program verifications using the current tooling and specifically how the existing verifications can be abstracted into a framework that serves a wide variety of use cases, while being convenient and flexible enough to be easily integrable into the ecosystem of Curry and FlatCurry tools. We will motivate these ideas in section 3.1, provide a step-by-step introduction to the *Application Programming Interface (API)* in section 3.2, formally discuss the semantics in section 3.3 and review the concrete architecture in section 3.4. Throughout the chapter, we will treat the framework as a black box, focusing on the interface between a verification tool and the framework, and leave any implementation decisions for later discussion in chapter 4.

3.1 Motivation

While the existing verification tools, notably including the contract prover and the non-failure verification discussed in chapter 2, already share many libraries around the handling of FlatCurry, writing new verification tools traditionally involves reimplementing similar patterns for traversing, analyzing and outputting verification results[1]. CASS, introduced in [HS14], provides a foundation for simpler analyses, but lacks support for the mutation of programs, as needed to insert dynamic run-time checks, and per-function I/O to integrate with external tools like SMT solvers. This motivates our primary design goal, namely to abstract over existing verification tools, extract common techniques and provide an underpinning that serves both existing and new verification tools well. To achieve this, we introduce a new framework, inspired by CASS, that is geared towards the implementation of more complex program verifications and lets us generalize these patterns in a way that retains the flexibility of implementing verifications as external programs.

3.2 Approach

At the highest level, the idea of the framework is to provide the infrastructure for fixed-point computations that associate a verification-specific result value with every function in the program to be verified. Depending on the specific verification, this value could take on several forms: The simplest verifications could associate a unit value and communicate its outcome purely in terms of having the verification succeed or throw an error message[2]. The contract prover, on the other hand, could, when ported to the framework, associate a list of pre- and postconditions with each function, along with a flag for every condition that states whether it could be verified statically. The non-failure verifier, which

[1]Throughout the thesis, we will use the terms verification *results*, verification *values* and verification *info* interchangeably. The term *info* follows the naming conventions from CASS and will be used mostly for the implementation whereas the term *results* will be used in higher-level discussions. In some contexts we will also use the term *info* as a singular noun, with e.g. *function info* referring to the verification result for a single function and *program info* referring to the set of all function infos for a program.

[2]Such verifications could be performed in a single pass and would technically not require a fixed-point iteration.

in its more advanced form is capable of inferring non-failure conditions, could associate an abstract call type with each function, thus showing how functions can be called to avoid failures. As these verification values vary a lot, as the previous examples show, the framework has to be polymorphic over the verification value.

Taking the perspective of the designer of such a framework, a key goal is having a simple API that accommodates these use cases. If we let our framework do the heavy lifting of traversing the program and focus on solely what the client has to provide to the framework, the simplest design one might attempt to use would be to have clients provide a single function that takes a FlatCurry function declaration and returns a verification value to associate it with:

```
vAnalyze :: FuncDecl -> a
```

This idea in fact corresponds to the `SimpleFuncAnalysis` provided by CASS, which has been used to implement simple program analyses that can be expressed as pure functions over the FlatCurry function declaration. Our verifications, however, require a number of additional effects, which we need to include in the signature. As mentioned above, verifications can fail and perform I/O, in order to e.g. call an SMT solver, therefore we introduce a new monad `VM` that captures these two effects:

```
type VM = ExceptT String IO
```

Using this monad, we can update the signature by wrapping the return type in `VM` to allow performing the corresponding effects:

```
vAnalyze :: FuncDecl -> VM a
```

It should, however, be noted that we consider I/O to be an implementation detail and still expect computations to be semantically pure. In other words, even though an external SMT solver may be called, we expect the function to return the same result for multiple invocations, including whether or whether not it errors. This will greatly simplify formal specification and analysis of the verification semantics, as we can treat the function as if it were pure without loss of generality.

Another requirement of many verifications is accessing other functions. In the case of the contract prover, for example, the verification of a function may have to access pre- or postconditions that are defined as separate functions. To allow this, we pass the surrounding FlatCurry program, i.e. a `Prog`, as an additional parameter[3]:

```
vAnalyze :: Prog -> FuncDecl -> VM a
```

This signature works for verifications where each function can be processed in isolation. In practice, however, it is common for verifications to depend on other verification values, usually the values of the function's callees. We express this by passing a map of the previous verification values to the function:

```
vAnalyze :: Map QName a -> Prog -> FuncDecl -> VM a
```

Note that the ordering of the calls to `vAnalyze` is now significant as it affects which previous verification values each function sees in the map. In a program without recursion, a topological ordering would be sufficient to ensure a function sees all of its callees' verification values, in practice, we need to handle recursion and mutual recursion too, however. The solution here is to split up the verification from a single pass to a multi pass process, where we perform an initial traversal over all functions to associate an initial value with them by calling `vInit` and then repeatedly iterate `vAnalyze` over all functions until the results no longer change or a maximum number of iterations has been reached:

[3]We intentionally do not include it in the `VM` monad to separate concerns: Not all functions need the full program and the client may wish to use `VM` in its own functions. If needed, the monad can always be wrapped in additional state via `StateT`.

```
vInit    ::                 Prog -> FuncDecl -> VM a
vAnalyze :: Map QName a -> Prog -> FuncDecl -> VM a
```

This effectively implements a fixed point iteration and resembles the `DependencyFuncAnalysis` from CASS, though our signature differs slightly in that we provide the program as context and allow additional effects via the `VM` monad.

The long function signatures are getting a bit unwieldly, so we move the `Prog`, `FuncDecl` and also the `Map QName a`, which we initialize with `Map.empty`, into a combined data type named `VFuncEnv` that we pass to `vInit` and `vAnalyze`. We will discuss the precise layout of these types later, for now the key idea is that `VFuncEnv` encapsulates the entire context around the current function to be verified, including its surrounding program and any previous verification values:

```
vInit    :: VFuncEnv a -> VM a
vAnalyze :: VFuncEnv a -> VM a
```

In some cases, not all functions should be verified, e.g. the contract prover may wish to skip verification of the pre- and postconditions themselves. To handle this, we have `vInit` return an optional value instead, with `Nothing` signifying that the function should be skipped:

```
vInit    :: VFuncEnv a -> VM (Maybe a)
vAnalyze :: VFuncEnv a -> VM a
```

As mentioned earlier, verifications should also be able to mutate the function, e.g. to insert dynamic run-time checks, so we return an updated version of the current function, a `FuncDecl`, along with a list of functions to be added to the program[4], i.e. a `[FuncDecl]`:

```
vInit    :: VFuncEnv a -> VM (Maybe a)
vAnalyze :: VFuncEnv a -> VM (a, FuncDecl, [FuncDecl])
```

Since the result type of `vAnalyze` is getting a bit unwieldly, we move the updated verification result of type `a`, the updated function and the added list of functions into a type named `VFuncUpdate`. As with `VFuncEnv`, we will discuss the exact definition of these types later as the actual type includes some additional implementation details, for now we treat them as abstract types:

```
vInit    :: VFuncEnv a -> VM (Maybe a)
vAnalyze :: VFuncEnv a -> VM (VFuncUpdate a)
```

While this already handles the iteration itself well, some verifications need to perform additional validation or preprocessing on the functions prior to the iteration. The contract prover, for example, ensures that pre- and postconditions are associated with a declared function per their naming scheme, e.g. that if a function like `f'pre` exists, a function named `f` has to be declared in the same module too. To handle this case and potentially other verification-specific transformations in a generic way, we let verifications additionally provide a `vPreprocess` function that provides an opportunity for both validation, by using the `VM` monad, which supports throwing errors, and transformation, by optionally letting the verification return a modified `Prog`:

```
vPreprocess :: Prog       -> VM (Maybe Prog)
vInit       :: VFuncEnv a -> VM (Maybe a)
vAnalyze    :: VFuncEnv a -> VM (VFuncUpdate a)
```

[4]The contract prover e.g. uses separate functions to encapsulate the dynamically checked and unchecked version of a function, along with providing the check itself as a separate function.

$$\begin{array}{llll}
\text{(Init)} & \Gamma : P \Rightarrow \Gamma[f \mapsto x] : P & \text{where } f \in P \setminus \Gamma, \\
& & \quad \texttt{Just}\ x = \texttt{vInit}(P, f) \\
\text{(Update)} & \Gamma[f \mapsto x] : P \cup \{f\} \Rightarrow \Gamma[f \mapsto x'] : P \cup \{f'\} \cup F & \text{where } f \in \Gamma, \\
& & \quad (x', f', F) = \texttt{vAnalyze}(\Gamma[f \mapsto x], P \cup \{f\}, f), \\
& & \quad (x', f', F) \neq (x, f, \{\})
\end{array}$$

Figure 3.1. Operational semantics of the fixed-point verification iteration

For implementation purposes, we also abstract the left-hand side to a new type named `VProgEnv` and the right-hand side, in a similar fashion, to `VProgUpdate`. This lets us provide additional context without changing the signature later:

```
vPreprocess :: VProgEnv a -> VM (VProgUpdate p)
vInit       :: VFuncEnv a -> VM (Maybe a)
vAnalyze    :: VFuncEnv a -> VM (VFuncUpdate a)
```

These functions illustrate the basic interface between the verification framework and an implementation by a verification tool consuming the framework. To simplify the terminology, we will call the consuming tool the *client* of the framework. Since the types only tell half of the story, implementing a semantically meaningful verification requires having a detailed model of how the functions will be invoked, which we will discuss in the following section.

3.3 Formal Semantics

As mentioned in section 3.2, the key idea is to run a fixed-point iteration over the program and its functions' verification values. Figure 3.1 depicts a formal description of this fixed-point iteration as small-step operational semantics in the form of inference rules. These rules use judgments of the form "$\Gamma : P \Rightarrow \Gamma' : P'$", describing a single step, which informally states that the program P in an environment Γ can derive an updated environment Γ' and an updated program P'. An environment Γ represents a partial mapping from functions[5] in P to verification values and uses the square bracket notation to add or update a mapping. More formally, we let $\Gamma[f \mapsto x]$ denote a new environment Γ' where $\Gamma'(f) = x$ and $\Gamma'(f') = \Gamma(f)$ for $f' \neq f$. For notational convenience, we identify a program P with the set of its functions, i.e. use the notation "$f \in P$" to mean functions declared in P and similarly "$f \in \Gamma$" to refer to functions with an associated mapping in Γ.

As for the use of `vInit` and `vAnalyze`, we make a few simplifying assumptions to make them easier to work with in a mathematical context, without incurring any loss of generality:

▷ We treat the functions as if they were pure. To see how, we begin by assuming that we operate on the evaluated representation of the `VM` monad, i.e. that the returned values are of the form `IO (Either String _)` instead of their transformer representation. Per the previous section we can safely ignore `IO` as we explicitly assume that the functions are semantically pure, even when using

[5]In the implementation, the mapping is from the qualified names identifying the functions rather than the functions themselves. For brevity, we will still use the notation $f \mapsto x$ to denote that the function f's identifier maps to the value x.

external tools such as an SMT solver. This leaves us with `Either String`. For notational convenience we assume that `Right` x is represented directly as x and that `Left` $\_$ is represented as an arbitrary unused symbol that would fail our formal pattern matching in the where clauses of figure 3.1, causing them to no longer apply and thus terminate the iteration.

- ▻ Instead of wrapping the arguments and results in `VFuncEnv` and `VFuncUpdate`, we use the equivalent unpacked representations as tuples. In other words, we expect `vInit` to take a program P and a function f as inputs, returning an initial `Maybe a`, and `vAnalyze` to take an environment Γ, a program P and a function f, returning a triple of an updated value of type `a`, along with an updated function f' and a set of functions F to add.

The semantics can be applied by starting with the empty environment $\Gamma = \{\}$ and a program P. The two rules from figure 3.1 can now be interpreted as follows:

- ▻ In the (Init) rule, an initial verification value is computed for every function f in P by invoking `vInit` with f and P. Unless the value is `Nothing`, in which case the function will be skipped for verification, the value is then added to the environment Γ by associating it with f.
- ▻ The (Update) rule describes the heart of the fixed-point iteration: For every function f in the environment Γ, the function and its existing verification value x are extracted from Γ. Then, `vAnalyze` is invoked with Γ, P and f to compute an updated verification value x, a potentially modified version of the function f' and a set of new functions F, which are to be added to the program. If $x \neq x'$, i.e. the verification value changed, if $f \neq f'$, i.e. the function changed or if F is non-empty, i.e. new functions were added, the (Update) rule applies and provides an updated environment and program. If neither the value nor the functions change, no rules apply anymore and the iteration stops.

A nice way of thinking about the small-step semantics is as a state machine: Each state is a pair of the form "$\Gamma : P$" and the semantics describe the transition relation between these states. In our case, we would have start and end states of the following form, given an initial program P:

$$\begin{array}{lll} \text{(Start)} & \{\} : P & \\ \text{(End)} & \Gamma : P' & \text{where } \Gamma \text{ is an environment, } P' \supseteq P \text{ and} \\ & & \forall f \in P \text{ where } \texttt{Just}\ \_ = \texttt{vInit}(P, f) : (\Gamma(f), f, \{\}) = \texttt{vAnalyze}(\Gamma, P', f) \end{array}$$

This characterization of an end state effectively follows directly from the (Update) rule, because we capture precisely those states where this rule no longer applies. It is worth mentioning that an end state is not guaranteed to exist if the (Update) rule is always applicable. This corresponds to a non-terminating iteration, i.e. the case in which a fixed-point does not exist. For example, if `vAnalyze` updates the value or adds new functions in every iteration, the computation recurses indefinitely and a fixed-point is never reached. This is intentional, as some verifications naturally result in infinitely large verification values. The classic example for this is the `infpos` function under non-failure verification:

```
infpos n | n > 0 = infpos (n - 1)
```

In the first iteration, its associated non-failure condition is $\top$, in the second iteration it is $n > 0$, in the third $n > 0 \land (n-1) > 0$ and so on. Since we do not intend to find an exhaustive condition or verification value in every case[6], we limit the fixed-point iteration to a client-configurable maximum number of iterations in the implementation.

[6]For many concrete verifications, e.g. non-failure or termination checking, this is also undecidable in general.

3.4 Library Architecture

More holistically, our verification framework is a library that clients can hook into by providing implementations of the verification functions `vPreprocess`, `vInit` and `vAnalyze`, as sketched out in section 3.4.1. The library-oriented design contrasts with CASS, which is primarily designed around usage as a unified command-line or server tool, but provides clients with more flexibility to implement e.g. a custom command-line interface. The client-provided implementations are called by the framework[7]. To kick off the verification, the client calls one of the entry points listed in section 3.4.2, after which the framework calls the client in stages as described in section 3.5. These implement the formal semantics given above, along with some surrounding logic to read, preprocess and write the FlatCurry programs.

3.4.1 Verification API

The `vPreprocess`, `vInit` and `vAnalyze` functions introduced in section 3.2 already closely resemble the actual API implemented by the framework. There are just two small details missing from their signatures: First, we abstract over the specific program type, to handle both typed and untyped FlatCurry. This is done by adding a type parameter `p` for the program type, replacing all instances of `Prog` in the function signatures given earlier, and another parameter `f` for the function declaration type, replacing all instances of `FuncDecl`. Although the function declaration type logically depends on the program type, Curry's type system currently lacks support for type families, which would be needed to express this statically[8], hence we need both parameters. To make the API convenient to use, we bundle the verifications functions in a record that represents a complete implementation of a verification and is easy to pass around for both the client and the framework:

```
data Verification p f a = Verification
  { vPreprocess :: VProgEnv p f a -> VM (VProgUpdate p)
  , vInit       :: VFuncEnv p f a -> VM (Maybe a)
  , vAnalyze    :: VFuncEnv p f a -> VM (VFuncUpdate f a)
  }
```

To reduce verbosity, we provide `T`- and `U`-prefixed type synonyms that are parameterized with the types representing type-annotated and untyped FlatCurry programs, respectively:

```
type TVerification = Verification TAProg TAFuncDecl
type UVerification = Verification   Prog   FuncDecl
```

where `TAProg` is the type of a program in type-annotated FlatCurry and `TAFuncDecl` the type of a function declaration. `Prog` and `FuncDecl` are the types of the equivalent respective constructs in untyped FlatCurry. A full overview of all FlatCurry types can be found in appendix A.

Analogously defined type synonyms are provided for all other verification types, following a similar naming scheme: `VTProgEnv`/`VUProgEnv`, `VTProgUpdate`/`VUProgUpdate`, `VTFuncEnv`/`VUFuncEnv`, and so on. For brevity, we will prefer these prefixed type synonyms in the context of untyped or FlatCurry and use the original type names when discussing general ideas that apply to both forms of FlatCurry.

[7]This design pattern, called *Inversion of Control (IoC)*, is often viewed as a defining characteristic of frameworks, as opposed to the usual structure of software libraries in terms of disjoint functionality [Fow05].

[8]In principle, MPTC and functional dependencies could also be used to model this by moving all FlatCurry related functions into a type class that `Prog` and `TAProg` implement. Unfortunately, Curry's support for MPTC with functional dependencies is not stable enough to make this work well enough in practice, so we choose to go with the more verbose option of carrying along the extra type parameter.

3.4.2 Entry Points

The entry points constitute are the top-level functions that perform the verification from the client's view. They constitute the main interaction with framework as they yield control for the remaining lifecycle of the verification. Corresponding to the split between type-annotated and untyped FlatCurry, the framework offers two entry points:

▷ `runTypeAnnotatedVerification` executes a **`TVerification`**, i.e. one that uses type-annotated FlatCurry

▷ `runUntypedVerification` executes a **`UVerification`**, i.e. one that uses untyped FlatCurry

Note that we do not provide an entry point that handles both untyped and type-annotated FlatCurry generically. This is a design decision as exposing the fully polymorphic interface would impose both a much more verbose interface on the client and leak some implementation-specific abstractions such as **`VProgHandlers`** that we will discuss later in chapter 4. We expect most concrete verifications to use one of the two formats, so we do not consider it to be a likely issue in practice.

Both entry points take the verification itself, along with a **`VOptions`** type, which contains all configuration that is provided to the framework, and return a **`VState`** representing the final state of all programs and verification values. While the full definition of **`VOptions`** can be found in appendix B, we will briefly review the most important ones, which are defined as follows:

```
data VOptions a = VOptions
  { voName               :: String          -- The name of the verification
  , voModules            :: [String]        -- Source paths of all Curry modules to verify
  , voLog                :: VMessage -> IO () -- A handler for log messages
  , voCacheEnabled       :: Bool            -- Whether caching is enabled.
  , voMaxIterations      :: Int             -- The maximum number of fixed-point iterations
  , voWriteProgs         :: Bool            -- Whether to write the transformed programs
  , voWriteUntypedProgs  :: Bool            -- Whether to write the transformed programs
                                            -- after conversion to untyped FlatCurry
  , ...
  }
```

▷ The `voName` option is the only mandatory one. It stores the name of the verification, which is used for pretty-printing and internal caching purposes.

▷ The `voModules` option contains the list of modules to verify. Each module may either be specified as a source path or as a module identifier, though the latter requires the module to be on the CurryPath, a standard mechanism also used by other Curry tools to resolve modules to source locations on disk. The CurryPath which may be customized in a variety of different way, including via the `CURRYPATH` environment variable or the `setCurryPath` function, though the details are handled by the `currypath` package.

▷ For logging, the framework provides a simple abstraction in the form of a log handler named `voLog`. The idea is relatively simple: All log messages produced by the framework will be routed through this handler, providing flexibility to the client in how these messages are to be handled. The default log handler, `noLog`, will simply discard any messages, but a `printLog` handler that writes to standard out, along with a combinator named `withVLevel` for setting the log level are provided by the framework too. The full logging API can be found in section B.2.

- ▻ To control whether caching of verification values is enabled, the options include a `voCacheEnabled` flag. We will discuss the precise details around this later in chapter 4, but it should be noted that this mechanism is optional.
- ▻ Following the ideas from section 3.3, we need a limit for the maximum number of fixed-point iterations during the verification. By default, our limit is 16 iterations, but we let the client configure via the `voMaxIterations` flag.
- ▻ Finally, two options are provided for controlling the write behavior of the framework. As motivated earlier, some verifications, like the contract prover, mutate the FlatCurry programs, adding dynamic checks or even new functions. Like other tools operating on FlatCurry, the framework writes the modified programs back to the path they were read from, as determined by the CurryPath, though the precise path depends on whether untyped or typed FlatCurry is chosen:
  - ▻ The `voWriteProgs` flag writes the programs in the same format as they were read.
  - ▻ The `voWriteUntypedProgs` flag, on the other hand, will write them as untyped FlatCurry, even when type-annotated FlatCurry is used internally. This is especially useful for verification tools that want to give their users control over this output format.

The concrete type signatures for the two entry points now look as follows:

```
runTypeAnnotatedVerification :: (Read a, Show a, Eq a)
                             => TVerification a -> VOptions -> IO (Either String (VTState a))

runUntypedVerification       :: (Read a, Show a, Eq a)
                             => UVerification a -> VOptions -> IO (Either String (VUState a))
```

As outlined earlier, the entry points take both the client-provided `Verification` and `VOptions`, both of which are parameterized over the verification value type `a`. This type parameter is constrained by `Read` and `Show` for serialization and caching purposes, along with `Eq` for diffing, as the framework has to be able to compare the values to determine changes during the fixed-point iteration. The result is wrapped in an `Either String` to handle potential errors, as well as `IO`, as both the framework and the implementation may perform I/O, e.g. to read/write files or invoke external SMT solvers. Note that we intentionally do not use the `VM` monad here as that monad is intended for e.g. throwing errors during the verification, which we wish to handle at this point.

3.5 Verification Flow

Putting these ideas together, figure 3.2 illustrates the call structure of a verification implemented using the framework, from the client's point-of-view. As discussed in section 3.4.1, the concrete types vary based on whether the entry point for untyped or type-annotated FlatCurry is chosen, hence the parameterization over the type parameters `p` and `f`. The general idea is the same for both: At the highest level, the client invokes the entry point using the `Verification` encapsulating the implementations of `vPreprocess`, `vInit` and `vAnalyze`, along with a `VOptions` value for configuration, notably including the list of modules to be verified. The framework then proceeds as follows:

- ▻ In the first phase, the framework loops over all modules, calling the client's `vPreprocess` to validate and potentially update each program, with the updated program being communicated via a returned `VProgUpdate`.

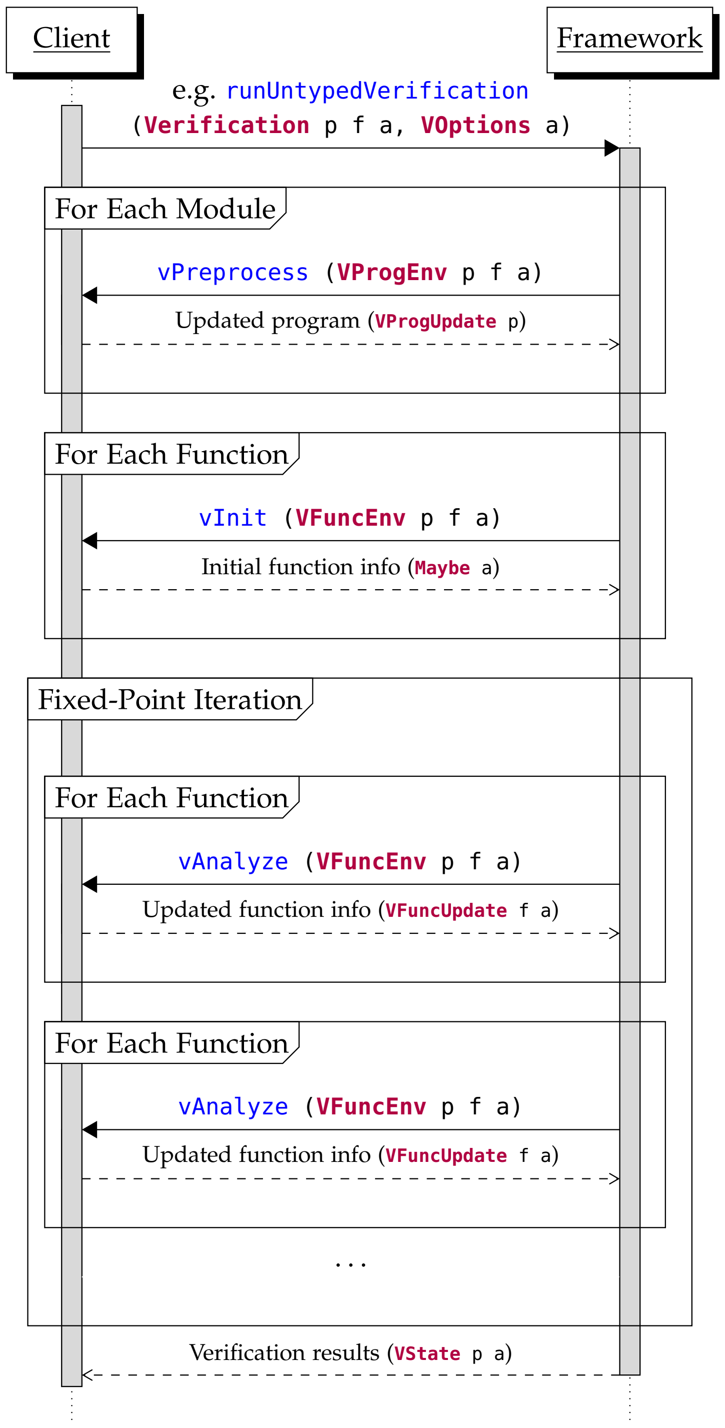

Client
Framework
e.g. runUntypedVerification
(Verification p f a, VOptions a)
For Each Module
vPreprocess (VProgEnv p f a)
Updated program (VProgUpdate p)
For Each Function
vInit (VFuncEnv p f a)
Initial function info (Maybe a)
Fixed-Point Iteration
For Each Function
vAnalyze (VFuncEnv p f a)
Updated function info (VFuncUpdate f a)
For Each Function
vAnalyze (VFuncEnv p f a)
Updated function info (VFuncUpdate f a)
...
Verification results (VState p a)

Figure 3.2. Abstract call structure of framework-based verifications

▷ In the second phase, the framework iterates over every tracked function, calling `vInit` with an environment that includes all global state and state about the current function, namely a **`VFuncEnv`**. In contrast to the preprocessing, this iteration includes both all functions from modules explicitly specified in the **`VOptions`** and any functions that they directly or transitively depend on[9]. The **`Maybe`** `a` returned from `vInit` by the client provides the initial verification value or, if **`Nothing`** is returned, removes the function from the set of tracked functions.

▷ The third and final phase includes the fixed-point iteration, which follows the semantics discussed earlier in section 3.3: In every iteration, the framework calls `vAnalyze` on every tracked function, obtaining a **`VFuncUpdate`** that describes a change to verification values or functions in the programs. The fixed-point iteration terminates once either no updates have occurred throughout an entire iteration or the maximum number of iterations has been reached. At the end, the framework returns a **`VState`** that includes all global state, notably the updated programs and verification values.

[9]There is a caveat to this, namely that it is not guaranteed that all dependencies are included when caching is enabled. This should, however, be considered an implementation detail that we will discuss later.

Chapter 4

Implementation

In this chapter, we will dive into the implementation of the verification framework, following the design presented in chapter 3. This includes both the internal module structure of the framework, which we will briefly review in section 4.1, an implementation view of the entry points in section 4.2, a discussion of state management within the framework in section 4.3 and a detailed overview of the verification runner in section 4.4, which is at the heart of the verification framework and implements the call structure outlined earlier in section 3.5. Additionally, we will discuss caching as a central optimization of the framework in section 4.5 and finally introduce some infrastructure for making verifications more scriptable, along with a tool for testing verifications on programs in a structured way, in section 4.6.

4.1 Package Structure

The verification framework is a Curry library named `verification` that uses the package format introduced by CPM. This makes it easy to integrate into Curry programs, as the majority of the Curry ecosystem has standardized on CPM packages, including the verification tools that inspired the framework, such as the contract prover or the non-failure verifier. By versioning the corresponding CPM package, it is easy to pin a specific version of the verification framework, a practice that has proven especially useful during development where the API may still undergo rapid and potentially even breaking changes. During this phase the framework will also use a zero-based major version, which in accordance with the common semantic versioning standard, allows any version to make breaking changes. Once the API stabilizes, we intend to synchronize the versioning with the remaining Curry ecosystem and maintain source-stability across major versions.

At the source level, the framework follows the common practice of keeping all modules under a common namespace based on the package name, in our case `Verification`. This aligns with other Curry packages, such as `flatcurry`, which prefer a short namespace unless the modules fit into a standard namespace, such as `Data` or `Control`, a convention inherited from Haskell. Figure 4.1 provides a full overview of the framework's modules. At a high level, the modules are organized around the type or function they represent and are grouped accordingly. For ease of consumption, most modules also re-export their submodules, so `Verification.Env`, for example, exports everything provided by `Verification.Env.Prog`. The general contract is that all modules except for `Verification.Internal` can be considered part of the public API and may be used freely by the client.

While a comprehensive reference of all modules can be found in appendix C, we will focus on the most relevant modules and introduce other modules only as needed. From a client-perspective, this includes the module hosting the public entry points, namely `Verification.Run`, and from an implementation-perspective also `Verification.Internal.Run`, where the majority of the verification runner is implemented. We will discuss both of these modules in more detail in the following sections.

```
Verification
├── Env                        -- Environment types
│   ├── Base                      -- VBaseEnv type
│   ├── Func                      -- VFuncEnv type
│   └── Prog                      -- VProgEnv type
├── FlatCurry                  -- FlatCurry-related helpers
│   ├── Annotated                 -- Annotated FlatCurry-related helpers
│   │   ├── Goodies                  -- Convenience functions
│   │   ├── Simplify                 -- Simplify implementation
│   │   └── Types                    -- Syntax types
│   ├── Goodies                   -- Convenience functions
│   ├── Simplify                  -- Simplify implementation
│   └── Types                     -- Syntax types
├── Handlers                   -- Abstraction over FlatCurry handlers
├── Info                       -- Info types
│   └── Prog                      -- VProgInfo type
├── Internal                   -- Internal modules
│   ├── Cache                     -- Cache types
│   ├── Monad                     -- VIM monad
│   ├── Run                       -- Verification driver implementation
│   └── Utils                     -- General utilities for the implementation
├── Log                        -- Logging API
├── Monad                      -- VM monad
├── Options                    -- Configuration and flags
├── Run                        -- Entry points (runUntypedVerification, etc.)
├── State                      -- Global state types
│   └── Prog                      -- VProgState type
├── Types                      -- Verification type
└── Update                     -- Update types
    ├── Func                      -- VFuncUpdate type
    └── Prog                      -- VProgUpdate type
```

Figure 4.1. Module tree of the framework package

4.2 Entry Points

As mentioned above, when looking at the implementation of the framework, the first module of interest is `Verification.Run`. This module implements the entry points mentioned in section 3.4.2, namely the functions `runUntypedVerification` and `runTypeAnnotatedVerification`, thus provides a key part of the framework's API. While the framework externally presents its entry points as two disjoint code paths, most of the framework's internals are polymorphic over untyped and type-annotated FlatCurry by using the `p` and `f` type parameters to abstract over the type of programs and function declarations as briefly outlined in section 3.4.1. Following this principle, both entry points delegate to an internal `runVerification` function that implements the verification runner generically, which we will later discuss in section 4.4. There is, however, still a missing piece of the puzzle, namely the question of how FlatCurry programs and functions are dealt with generically within the framework. The answer is the so-called `VProgHandlers` type, which encapsulates all mappings between different FlatCurry types in a single place:

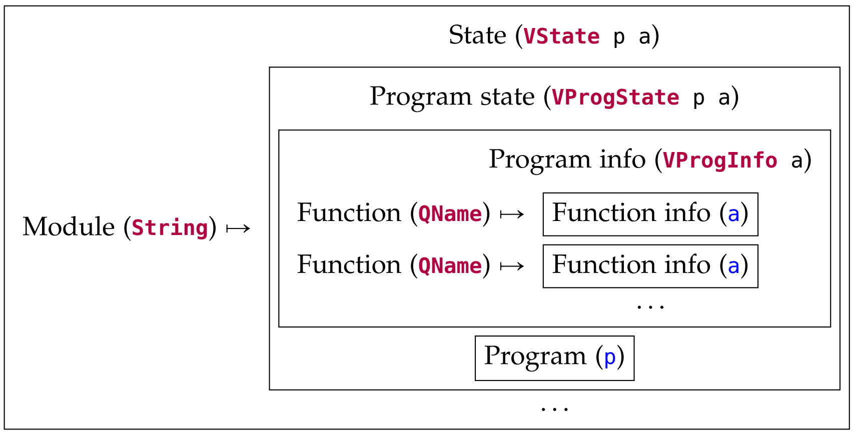


Figure 4.2. State hierarchy in the framework

```
data VProgHandlers p f = VProgHandlers
  { vphReadProg     :: String -> IO p     -- Read a program from disk, given a module name
  , vphWriteProg    :: p -> IO ()         -- Write a program to disk, at its canonical path
  , vphFuncName     :: f -> QName         -- Fetch the qualified name of a function declaration
  , vphFuncDeps     :: f -> [QName]       -- Fetch all function/constructor names in the body
  , vphProgFuncs    :: p -> [f]           -- Fetch all function declarations in the given program
  , vphProgTypes    :: p -> [TypeDecl]    -- Fetch all type declarations in the given program
  , vphSetProgFuncs :: [f] -> p -> p      -- Update the function declarations in the given program
  , vphSimpProg     :: p -> p             -- Simplify the given program
  , vphSimpFuncDecl :: f -> f             -- Simplify the given function declaration
  }
```

The functions largely correspond to the various accessors and helpers around FlatCurry known from `FlatCurry.Goodies` and `FlatCurry.Annotated.Goodies`, along with some other modules such as `FlatCurry.FilesRW` for file handling, and provide a generic interface to FlatCurry.

4.3 State Management

Before we discuss the runner, a crucial aspect of the implementation is how state is managed within the framework and how it is exposed to the client. Given that the fundamental premise of the framework is to coordinate the computation of verification results from programs, state is inherent to the task of verifying programs and managing it in a structured manner requires some careful consideration of different abstraction boundaries as the classic approach of managing it in an ad-hoc manner via `IORef`s, as commonly used in existing verification tools, does not scale well to a framework-based architecture.

4.3.1 Hierarchical Organization

Concretizing the ideas presented in chapter 3, figure 4.2 depicts how state is organized hierarchically within the framework: We distinguish between *function infos* of type `a`, i.e. the computed verification result for some function, *program infos* of type `VProgInfo a`, aggregating all function infos within a program, *program states* of type `VProgState p a`, which additionally stores the program itself, using the abstract type p, i.e. either Prog or AProg, depending on whether untyped or type-annotated FlatCurry is used, and finally *state* of type VState p a, which captures the global state across all modules.

Since we take a pure approach to state management, these types effectively describe an immutable snapshot of the state at a specific point during the verification. This is useful as it lets us deconstruct and pass around as much or little state as we want to, while requiring all updates to be passed explicitly, which stands in contrast to e.g. allowing interior mutability using IORefs.

4.3.2 Internal Monad

A key idea of the framework is to separate these update mechanisms, both to maintain a clear abstraction boundary between the framework's internals and the client and to enable fine-grained change tracking in the fixed-point iteration driving the verification. Internally, the framework uses the so-called VIM monad, which stands for *Verification-Internal Monad* and is defined as follows:

```
type VIM p f a = StateT (VIState p a) (ReaderT (VIEnv p f) VM)
```

Here, VIState tracks the internal read/write state of the framework. Currently, this is just a newtype around VState, but other state may be added in the future as needed:

```
newtype VIState p a = VIState { visState :: VState p a }
```

The more interesting type, VIEnv, is the internal read-only environment:

```
data VIEnv p f = VIEnv
  { vieOptions      :: VOptions
  , vieProgHandlers :: VProgHandlers p f
  }
```

The VOptions track the client-provided options, as described in section 3.4.2, whereas VProgHandlers include generic FlatCurry interface introduced earlier.

4.3.3 External Interface

The external interface defines how this state is exposed to the client. As already sketched out in section 3.4.1, we use so-called environments and updates to communicate inputs and outputs in the various Verification functions, respectively, deliberately hiding the VIM monad from the client. Aside from making the API more explicit, being able to inspect the updates requested by the client to functions and their verification info is crucial to the fixed-point iteration as it relies on tracking these changes, per the semantics described in section 3.3.

Environments

Environments encapsulate *inputs* to the functions provided in a Verification and come in three flavors: VBaseEnv for global context, VProgEnv for program-specific context and VFuncEnv for function-specific context. While the general idea corresponds to the design outlined in section 3.2, a key principle of the implementation is that these environments are *layered*: Each VFuncEnv includes a VProgEnv for the surrounding program, which in turn includes a VBaseEnv. This way we can simply choose the most specific environment for each purpose to include all contextual information.

The smallest of these environments, VBaseEnv, is defined as follows:

```
data VBaseEnv p f a = VBaseEnv
  { vbeOptions      :: VOptions a
  , vbeProgHandlers :: VProgHandlers p f
  , vbeState        :: VState p a
  }
```

Like `VIEnv` this environment includes the options and program handlers, here we include the `VState` from `VIState` too, however. In other words, we deliberately expose the internal read/write state as read-only to the client to enforce the idea that all mutations are performed by returning updates rather than through uncontrolled mutation of global state, which could break invariants around the fixed-point iteration and would not allow us to track changes as easily.

As mentioned above, the program environment `VProgEnv` is a wrapper around `VBaseEnv`:

```
data VProgEnv p f a = VProgEnv
  { vpeBaseEnv  :: VBaseEnv p f a
  , vpeModule   :: String
  }
```

The only addition to `VBaseEnv` is the name of the current module. Since we already track all programs and associated verification info in the `VState`, we can derive everything from there and provide a number of convenience functions on `VProgEnv`, e.g. for retrieving the current `VProgState`, the program itself or the function infos.

The function environment `VFuncEnv`, similarly, wraps `VProgEnv` to provide function-level context:

```
data VFuncEnv p f a = VFuncEnv
  { vfeProgEnv  :: VProgEnv p f a
  , vfeFunc     :: f
  }
```

For practical reasons we store the function declaration directly here, including to avoid linear time traversals over the corresponding program every time information about the current function is requested. Like with `VProgEnv` we provide a number of convenience functions to quickly fetch the current function's name or verification info.

Updates

Updates, on the other hand encapsulate *outputs* of `Verification` functions that are intended to trigger a change to program or function state. These come in two flavors, `VProgUpdate` for a program update during preprocessing and `VFuncUpdate` for a function update during the fixed-point iteration.

The program update `VProgUpdate` is effectively just a wrapper around the new program that a client may provide to the framework during `vPreprocess`:

```
newtype VProgUpdate p = VProgUpdate { vpuUpdatedProg :: Maybe p }
```

Unlike the design presented in section 3.2, we additionally wrap this new program in a `Maybe`. This is a purely practical decision as this lets us define an empty default update and lets us avoid mutating the state if the program does not change. Semantically, returning a `VProgUpdate Nothing` is equivalent to returning an update containing the current program, however, like we would have to in the formal semantics from section 3.3.

The function update `VFuncUpdate`, which is returned from `vAnalyze`, encapsulates a fixed-point iteration update. We define this type as follows:

```
data VFuncUpdate f a = VFuncUpdate
  { vfuUpdatedFunc :: Maybe f
  , vfuUpdatedInfo :: Maybe a
  , vfuAddedFuncs  :: [f]
  }
```

Here, the `vfuUpdatedFunc` represents the potentially mutated version of the function, `vfuUpdatedInfo` represents the computed verification info and `vfuAddedFuncs` any functions that are to be added to the program. Like with `VProgUpdate`, we apply the same optimization over the formal semantics here by taking `Maybe` values to simplify the internal tracking of deltas in the verification runner and to provide an empty `VFuncUpdate` as a convenient default.

4.4 Verification Runner

As mentioned before, the verification runner is at the heart of the framework. It orchestrates the execution of the client-provided verification, including the resolution of modules, preprocessing, simplification and fixed-point iteration, thus coordinating nearly all aspects of the framework centrally. The runner implements the call structure outlined in section 3.5 and is located in `Verification.Internal.Run`, with the main function being `runVerification`.

4.4.1 Overview

The role of the runner is analogous to that of a driver in traditional compiler architectures [ALS+07], performing a sequence of stages that begins at reading a program and ends at writing a transformed version of that program. Figure 4.3 provides a graphical overview over the runner's pipeline and shows at which points the client, i.e. the implementation of `vPreprocess`, `vInit` and `vAnalyze`, is called.

The signature of the runner's entry point `runVerification` is relatively straightforward, as it effectively just takes a client-provided `Verification` and executes it generically within the `VIM` monad:

```
runVerification :: (Show a, Eq a) => Verification p f a -> VIM p f a ()
```

Internally, this function drives the verification runner and invokes the stages in order.

4.4.2 Stages

The implementation of the runner's stages corresponds to the sequential architecture shown in figure 4.3. In the following section, we will take a closer look at these stages, both in terms of their precise role and how they are implemented.

Read Programs

In the first stage, the framework takes the list of module names provided via the `VOptions` and reads the corresponding FlatCurry programs[1] from the standard Curry path, which primarily includes the `.curry` folder in the source directory. Depending on which entry point is picked, either untyped (`.fcy`)

[1]As a short note on terminology: We will use the term *module* to refer to the abstract grouping of Curry types and functions and *program* to the concrete FlatCurry syntax tree. This distinction can get a bit fuzzy at times, but conventionally we consider *module* to be the more general term.

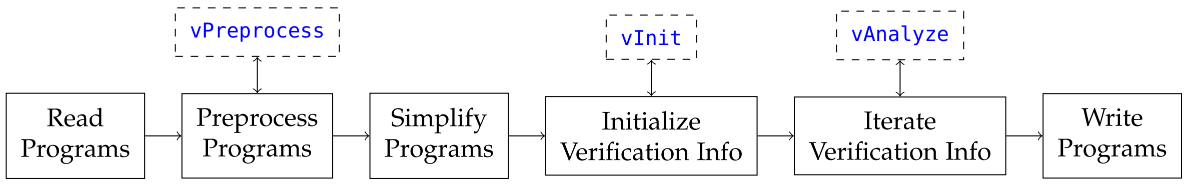


Figure 4.3. Flow structure of the verification runner

or type-annotated (`.afcy`) FlatCurry files are read[2]. Since Curry modules have dependencies, we need to recursively include the imports of the specified modules in our verification too. Traversing the module graph is a simple *Depth-First Search (DFS)* that reads and stores the FlatCurry program for every visited module and computes the topological order of the discovered modules.

An optimization of our framework is tracking dependencies at the function level to avoid unneeded overhead, especially in the case of large modules like the `Prelude`. To do this, the DFS traverses the callees of the visited functions in each module, grouping them by their module of origin, as opposed to e.g. traversing the module's imports directly. For every module, we store the discovered functions in a set of so-called *tracked* functions associated with that module. The modules explicitly passed to the framework's entry point implicitly track every function while their dependencies only track the functions that are actually used downstream. To illustrate this with an example, consider the following program:

```
x :: Int
x = head [1, 2, 3]
```

The right-hand side of `x`, after desugaring the list literal syntax, depends on the list constructors `(:)` and `[]`, as well as the `head` function, all of which are declared in the `Prelude`. The `head` function, in its desugared FlatCurry representation, also depends on `failed`. Thus, our set of tracked functions for the `Prelude` is `{head, failed}`, given that constructors do not have to be verified, and the set of tracked functions for our analyzed module is the set of all its functions, in this case just `{x}`.

Preprocess Programs

Once the program for every specified and, possibly transitively, included module has been read, the client is given the chance to perform preprocessing, as motivated earlier in section 3.2. The preprocessing stage operates on both the modules explicitly specified in `VOptions`, as well the dependencies discovered in the Read Programs stage. This applies to all further stages, except for Write Programs, which we will discuss later. In other words, all dependencies are fully included in the verification, enabling verifications to span multiple modules.

This stage is the first one to invoke the client by calling `vPreprocess` from the `Verification` on all modules. This gives the client the chance to both perform preliminary checks, possibly bailing out early with an error message, and to modify the program itself. Notably, the modules are only traversed once and in an unspecified order, so any preprocessing done by the client has to be local to each module and not affect other modules like the verification itself may.

While preprocessing is an inherently verification-specific task, contract usage checking is a simple example for a preprocessor that performs validation without changing the program. Consider a program of the following form:

[2]Internally, the `readFlatCurry` function takes care of automatically compiling the source files with the Curry frontend, thus neither the client nor the user have to do this separately.

```
f'pre  :: Int -> Bool
f'post :: Int -> Int -> Bool
g'post :: Int -> Int -> Bool
f      :: Int -> Int
```

The usage checker will first check `f'pre` and `f'post`. Both of these contracts reference a function `f`, which exists, thus pass the check. However, since there is no function `g`, the check will fail once it reaches `g'post` and emit a corresponding error message.

Simplify Programs

While the previous stage takes care of verification-specific preprocessing, this stage applies generic transformations to each module and is fully implemented within the framework. Since program verification via generated SMT assertions plays a key role in many verifications, including the contract prover or the non-failure verifier, the central idea behind this stage is to simplify a number of common constructs into an easily translatable form and thus reduce the chance that the SMT solver will run into an unsolvable edge case. To illustrate this, consider this relatively simple program:

```
x :: Int
x = 3 + 4
```

The direct translation to FlatCurry looks roughly as follows[3]:

```
f(x) = _impl#+#Prelude.Num#Prelude.Int#(3, 4)
```

The translated function calls into the `(+)` implementation from the **`Num Int`** instance from Curry's **`Prelude`**. Inspecting the FlatCurry translation of the **`Prelude`**, however, shows us that we cannot just include these functions in the SMT translation as they are declared **`external`** and thus handled in the Curry runtime:

```
_impl#+#Prelude.Num#Prelude.Int# :: Int -> Int -> Int
_impl#+#Prelude.Num#Prelude.Int#(v1, v2) = plusInt(v1, v2)

plusInt :: Int -> Int -> Int
plusInt(v1, v2) = (prim_plusInt $# v2) $# v1

prim_plusInt :: Int -> Int -> Int
prim_plusInt external
```

We solve this by renaming all function calls during the simplify stage according to a list of client-provided unary and binary operations that are to be considered primitive. Making these configurable decouples the SMT representation from the framework's internals and lets the framework handle the renaming in a generic manner. In the API, the function name mappings are represented as two lists of pairs, for unary and binary operations respectively, and included in the client-provided **`VOptions`**:

```
data VOptions a = VOptions
  { ...
  , voUnaryPrimOps  :: [(String, String)]  -- Unary primitive operation name mapping
  , voBinaryPrimOps :: [(String, String)]  -- Binary primitive operation name mapping
  , ...
  }
```

[3]Unlike in section 2.1.5 we intentionally spell out the operator implementation here as this distinction is relevant here.

SMT-based verifications like the contract prover generally use the mappings defined in the `flatcurry-smt` package. In the specific example given before, the `_impl#+#Prelude.Num#Prelude.Int#` function from FlatCurry would be mapped to its corresponding SMT representation, a simple `+`:

```
binaryPrimOps :: [(String, String)]
binaryPrimOps = [..., ("_impl#+#Prelude.Num#Prelude.Int#", "+"), ...]
```

Of course, clients are still free to configure a different mapping or omit the operations entirely to keep the FlatCurry names and effectively skip the renaming.

Aside from the renaming of primitive operations, there are a number of other simplifying transformations that largely have also been factored out from existing verification tools like the contract prover. These are aimed at reducing the burden of e.g. translating to SMT later, as they effectively normalize some common patterns to a standard form:

▻ Folding nested partial applications into a single call with multiple arguments. For a binary operation `op`, this amounts to `apply(apply(op, x), y)` $\mapsto$ `op(x, y)` for expressions `x` and `y`.

▻ Transforming common `Prelude` functions such as `(?)` or `($)` into their direct FlatCurry equivalents, i.e. `x ? y` $\mapsto$ `x or y` and `f $ x` $\mapsto$ `f(x)` for an unary function `f` and expressions `x` and `y`.

▻ Folding dead case branches, e.g. when matching on a constant like `otherwise`. The canonical example here is `case otherwise of { True -> x; False -> y }` $\mapsto$ `x`.

In the stage's implementation, the expressions in every tracked function are traversed and repeatedly simplified until no transformation rule applies. It should be noted that every transformation is local to the syntax node it applies to, hence the heart of the simplify stage consists of a set of functions that recursively traverse the syntax tree and apply the corresponding transformations:

```
simpProg     :: SimplifyEnv -> Prog     -> Prog
simpFuncDecl :: SimplifyEnv -> FuncDecl -> FuncDecl
simpExpr     :: SimplifyEnv -> Expr     -> Expr
```

where `SimplifyEnv` encapsulates the client-provided `voUnaryPrimOps` and `voBinaryPrimOps`.

Initialize Verification Info

This stage marks the beginning of the actual verification and implements the (Init) rule from the semantics given in section 3.3. Like in the previous stages, we traverse all tracked functions in a single pass, this time, however, we invoke `vInit` from the client-provided `Verification` to compute an initial verification value for each function. As outlined earlier in section 3.2, the actual value is highly verification-specific and can take on different forms, depending on the desired output of the verification. Just to give a few examples:

▻ For the contract prover, the value is roughly[4] of the form `([Cond], [Cond])` with `Cond` roughly being a `(String, Bool)`, thereby recording a list of found pre- and postconditions where each condition has a name and a flag whether it could be verified successfully.

▻ For the non-failure verifier, the value is structurally equivalent to a triple of the non-failure condition, call type and I/O type, namely `(Maybe NonFailCond, Maybe (ACallType a), Maybe (InOutType a))` where the type parameter `a` represents the term domain, i.e. an abstract set of data terms.

[4]The actual types used in the contract prover are records, we use tuples here just to quickly showcase the structure of these types. We will discuss the specifics later in section 5.1.1.

Depending on the verification, there are many possibilities for what this stage could do[5]. In general, it is a good place to do preparatory processing that operates on the verification value and to assign defaults. One should not be deceived by the name, however, as the preparatory processing can get more complex too, as is the case in the non-failure verifier, which has to derive the initial call types via the scheme specified in [Han24] in this stage.

There is a subtlety in this stage's semantics, that is not to be overlooked: As specified in the semantics from section 3.3, the `vInit` function defines its return type as `Maybe a`, given the client the opportunity to filter out functions from the subsequent verification iteration by returning `Nothing`. This is useful for e.g. skipping the pre- and postconditions themselves during the verification, as they are already handled via their associated functions, but also requires every verification value to have a reasonable default. In practice, this should not be an issue, however, as the client can always decide to wrap their verification value in additional `Maybe`s as needed.

Iterate Verification Info

After every function has been assigned an initial verification value, the core verification takes place: We first iterate over all tracked functions that have an initialized verification value[6] and invoke the client-provided `vAnalyze` function on each tracked function. This returns an updated verification value and optionally a modified version of the function, along with a list of new functions to be added. We compare the state after the update with the state before and if there are any changes, we run another iteration until either the `voMaxIterations` limit from the `VOptions` is reached or there are no changes for any functions. As a memory optimization, we also provide an option named `voEnforceNF` which, when set to `True`, evaluates all verification values to NF after computing them.

This corresponds to the fixed-point iteration as described by the (Update) rule in the formal semantics from section 3.3, with the caveat that we need the limit to avoid looping endlessly on some functions. For example, inferring the non-failure condition of a function like `infpos` would add a new clause in every iteration, thus we need to place a bound on the number of iterations to ensure the verification terminates.

The ability to mutate the program by either changing the current function or adding new functions is useful as it allows the verification to add fallback dynamic checks when e.g. generated SMT assertions cannot be statically verified. The contract prover, for example, uses this to enforce preconditions by renaming the function to include the `'NOCHECK` suffix and then adding function with the original name, implemented as a wrapper around this unchecked version enforcing the precondition. To illustrate this, consider the following program using the integer factorial function `fac` discussed earlier:

```
f :: Int -> Int
f x = fac x
```

The contract prover will query the SMT solver, but be unable to prove that the call to `fac` in `f` satisfies the precondition and thus replace the call with a wrapper around `fac` that checks the precondition at run time. The FlatCurry code generated for this example, including functions added by the contract prover, looks as follows:

```
fac(v1) = checkPreCond(_inst#Prelude.Show#Prelude.Int#, fac'NOCHECK(v1), fac'pre(v1), "fac", v1)
```

[5]Simple one-shot verifications could even do the entire processing in this stage and leave the Iterate stage empty. Most verifications, however, need the fixed-point iteration to deal with mutually recursive functions, thus we will mostly focus on those that do perform both initialization and iteration.

[6]Functions that have not been assigned a verification value in the Initialize Verification Info stage are thus skipped.

```
fac'NOCHECK(v1) = case (v1 == 0) of
                    True -> 1
                    False -> case (v1 > 0) of
                      True -> v1 * fac'NOCHECK(v1 - 1)
                      False -> failed

fac'pre(v1) = v1 >= 0

f(v1) = fac v1

checkPreCond(v1, v2, v3, v4, v5) = case v3 of
                                     True -> v2
                                     False -> error $ ("Precondition of operation \'" ++ (v4
                                       ++ ("\' not satisfied for arguments:\n"
                                         ++ show(v1, v5))))
```

As we can see, the `fac` implementation has been renamed to `fac'NOCHECK` and replaced with a new function that checks the condition, for which another helper function, `checkPreCond`, has been added[7].

For our verification iteration this raises a few interesting questions. First we have to define when the mutation should happen. We could defer the mutation to after verifying all functions by aggregating any new functions added along the way, applying the changes first before continuing with the next iteration of the fixed-point loop, or just mutate the program directly. The latter approach could interfere with the traversal over all functions, particularly if we use some form of indexing, hence the first approach is the one we will adopt for our framework. Secondly, and perhaps more interestingly, it is not obvious whether any newly added functions should be considered for (re)verification in following fixed-point iterations. In the case of dynamic checks, as e.g. added by the contract prover, it generally does not make much sense to do so, since the checks are already deemed "safe" and reverifying them after mutating the program would risk trapping the fixed-point computation in an infinite loop of adding new precondition checks. On the other hand, our approach of "ignoring" any updated functions may yield incorrect results if the specific verification relies on computing a fixed-point over mutations to the function itself. Since we expect this to be uncommon in practice and design the framework primarily with the dynamic run-time check in mind, we will stick with the simpler approach of leaving any newly added functions out of the fixed-point loop and not triggering a new iteration when a function is mutated.

Write Programs

Finally, the transformed programs are written to disk for all modules explicitly specified in the `VOptions`, provided that the `voWriteProgs` or the `voWriteUntypedProgs` flag is set. If the entry point for type-annotated FlatCurry is used, the former flag will write the modules in the same format, i.e. as `.afcy` files, and the latter flag as untyped FlatCurry, i.e. as `.fcy`. If the entry point for untyped FlatCurry is used, both flags have the same meaning and setting either one of them will cause the modules to be written as untyped FlatCurry[8]. Note that the list of modules to be written does not

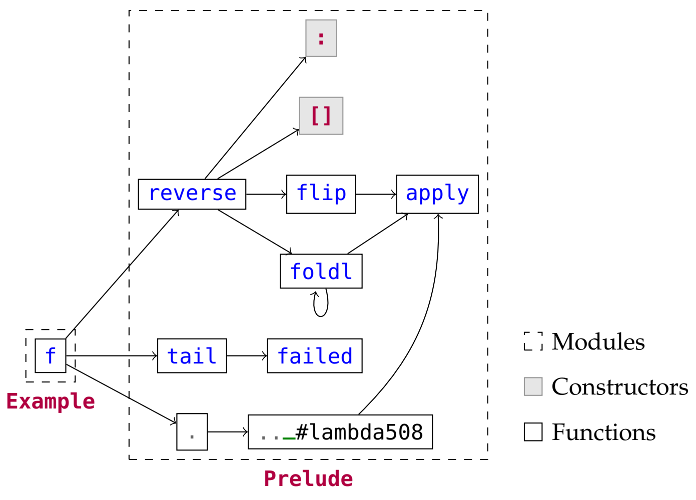
[7]Postconditions are handled similarly, for these a `checkPostCond` function is generated and invoked around the function's implementation. The precise details are not too important here as we focus on the more general mechanism underlying this example instead.

[8]We cannot provide a flag for the reverse case, i.e. writing type-annotated progs when parameterizing the framework over untyped FlatCurry as there are no type annotations to use. The conversion from type-annotated FlatCurry to untyped FlatCurry is thus inherently a lossy one.

```
module Example where

f :: [a] -> [a]
f = reverse . tail . reverse
```

(a) Curry source code



(b) Function dependency graph

Figure 4.4. Function dependencies in an example program

include the implicitly included dependencies, since we generally do not overwrite any FlatCurry modules not explicitly passed to the framework to avoid e.g. updating compiled standard library modules such as the `Prelude`, which may even be located in a read-only location if the Curry system was installed via a system-level package manager.

4.5 Caching

A central optimization of the verification framework is *caching*, a technique for memoizing previous results, thus making verifications incremental across executions. We will go further into the motivation behind introducing caching, however the fundamental principle is that it not change the semantics of verifications[9] and thus purely optimizes the architecture presented in the previous sections.

4.5.1 Motivation

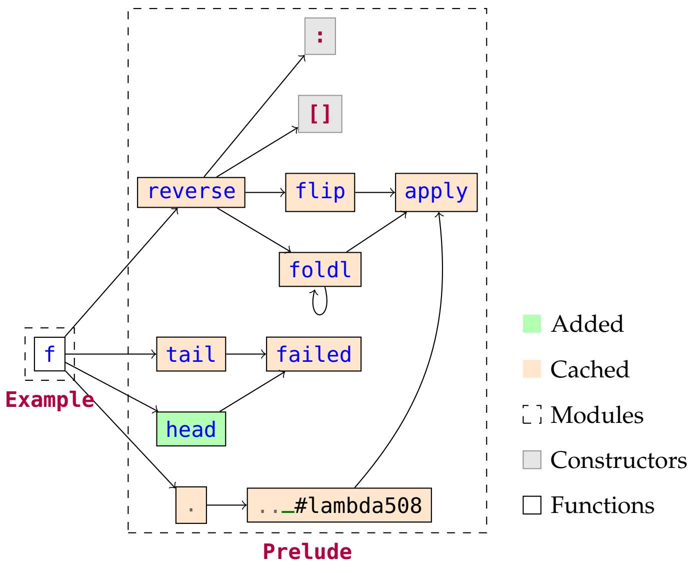
Even with all the verification infrastructure from section 4.4 in place, verifying programs can take a while in practice. Every Curry program already depends on a relatively large module, namely the `Prelude`, which has to be to read in full. Additionally, the number of included modules often grows as a Curry program scales. Though our framework already performs fine-grained function-level dependency resolution to ensure that only the functions used by the program are included in the verification, both the I/O around parsing thousands of lines of code and the number of functions included in the verification quickly become bottlenecks in practice. Figure 4.4 illustrates how a simple example program already has a number of direct and transitive dependencies: The function `f` depends on `reverse`, `tail` and the composition operator `(.)`, each of which have their own dependencies within the `Prelude`. Recursive functions, such as `foldl`, are particularly interesting as they may prolong the verification with additional fixed-point iterations. All of this is on top of the time it takes to read and

[9]If caching did change the program's semantics, that would be a bug in the framework!

```
module Example where

f :: [a] -> a
f = head . tail . reverse
```

(a) Curry source code



(b) Function dependency graph

Figure 4.5. Cached dependencies in modified example program

parse the **Prelude**, which especially with interpreted Curry backends such as PAKCS, where I/O is frequently a bottleneck, noticeably delays the verification.

4.5.2 Approach

Caching proves to be an efficient solution to these problems that both existing verification tools like the non-failure verifier or CASS have adopted before. Before looking into the implementation, there are, however, two big questions that need to be answered: First, we have to specify what we want to cache. Secondly, we need to figure out how to store it and how to invalidate the cache without changing the semantics of the verification. At the surface-level, the answers are relatively simple: We wish to persist the verification values and invalidate them whenever the program or its dependencies change. Both of these questions, however, have subtleties that require clarification.

One goal of caching is to avoid reading the full program of a module dependency if we can resort to cached verification info instead. Doing so without breaking the existing semantics is not entirely trivial though: When implemented as described in section 4.4.2, the Read Programs stage has to read and parse the program for every module dependency to walk the entire graph of function dependencies. If we were to naively cache the verification values, then read the cache instead of traversing the program, the Read Programs stage would be unable to discover any dependencies of the cached module. The framework has to know the full function dependency graph for two reasons: First, it is the only way to guarantee that the cache is semantically valid and that no upstream dependencies have changed, i.e. are *dirty*, and could thus invalidate the downstream module. Secondly, the environments exposed to the client have to contain all tracked functions in order for caching to be completely transparent to the client. If a module were to be missing because it was cached, that would be a side effect observable to the client, which is something we wish to avoid.

The solution that we use is to cache not just the verification info itself, but also the function dependency graph. Given that this graph is inherently much smaller than the program itself, we can both check whether the functions we are looking for during the Read Programs traversal is covered by the cache and traverse the dependencies of cached functions quickly without reading the program. This also nicely extends the principle of only verifying functions on-demand, instead of on a whole-module-basis, as only tracked functions are appended to the cache. Figure 4.5 shows how modifying the example from figure 4.4 affects this graph and how large parts of the existing info, namely the modules marked "Cached" in the graph, can be reused.

4.5.3 File Structure

The verification framework, by default, uses a global cache located in `~/.curry_verification_cache`, in analogy to the `~/.curry_analysis_cache` directory used by CASS. This has the advantage of being able to share cached info across projects, which is especially useful for heavily reused dependencies, such as the `Prelude`. To accommodate other use cases, the framework provides a way of configuring the path, however, by setting `voCacheRootDir` in the `VOptions`. Under the root directory, the framework, by default, places everything under the path `v1/<currysystem>`, e.g. `v1/pakcs-3.8.0` when using PAKCS 3.8. The version number is used as an additional cache key that the framework should bump whenever the persisted schema is changed in a backwards-incompatible way. This subdirectory, however, is also configurable and can be overridden via the option `voCacheSubdir`. We add another directory level under this path named after the verification, which can be specified using `voName`, and optionally additional `voCacheKeys`, all of which will be concatenated to a hyphen-separated directory name. For example, the cache directory for the contract prover, assuming it uses the name `ContractProver`, would be located at the following path:

```
~/.curry_verification_cache/v1/pakcs-3.8.0/ContractProver
```

Within this directory, the info for every cached module is stored under a path corresponding to that module's location in the file system. For example, the `Prelude`'s source code might be located at `/opt/pakcs/lib/Prelude.curry` for a PAKCS installation at `/opt/pakcs`, in which case the corresponding verification caches would be located under

```
~/.curry_verification_cache/v1/pakcs-3.8.0/ContractProver/opt/pakcs/lib
```

This approach is also borrowed from CASS and provides a simple way of identifying a module uniquely without risking name collisions. As outlined in section 4.5.2, the framework stores two cache files per module, namely a map of function names to their dependencies of the form:

```
newtype VProgDepsCache = VProgDepsCache { vpdcFuncDeps :: Map String [QName] }
  deriving (Show, Read, Eq)
```

This type is stored in its `Show` representation under `<module name>.deps`. The other file stores the set of tracked functions, along with the verification info:

```
data VProgStateCache a = VProgStateCache
  { vpscTrackedFuncs :: Set QName
  , vpscInfo         :: VProgInfo a
  }
  deriving (Show, Read, Eq)
```

Also using its `Show` representation, the prog state cache is stored under `<module name>.state`. For the `Prelude` we thus get `Prelude.deps` and `Prelude.state`. In the framework, the corresponding type definitions listed above can be found under `Verification.Internal.Cache`.

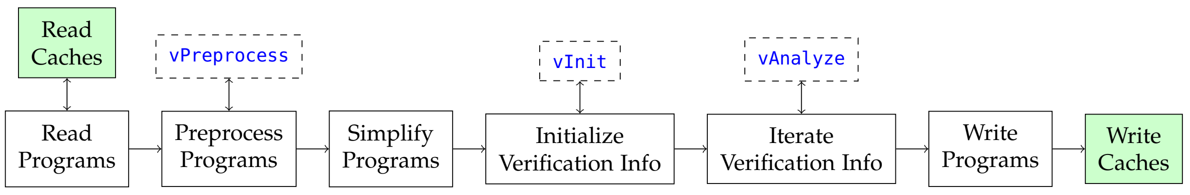


Figure 4.6. Extended structure of the verification runner with caching

4.5.4 Integration

We implement the caching mechanism described in section 4.5.2 by extending the Read Programs implementation to validate and read the cache files. This happens in two steps: First, we update the implementation of the DFS that resolves the tree of function dependencies to add a check for whether an up-to-date `.deps` file exists before reading a module. If the file exists and has not been invalidated, we skip reading the module and continue the traversal with the cached dependencies, otherwise we read the FlatCurry program as normal, parsing the function dependencies from the function declarations. In the second step, we read the `.state` files for all tracked modules. Note that we make the assumption that if the `.deps` file is up-to-date, the `.state` file is too. Unless the cache directory is corrupted, this assumption should hold in practice as both cache files are written together and are expected to have nearly the same timestamps.

The rationale for doing the heavily lifting of validating and reading the caches within Read Programs rather than a second stage stems from the observation that the caching logic is inherently intertwined with the read logic, especially with the ability to mix-and-match cached and uncached modules as the dependency graph is discovered on-the-fly during the DFS. Additionally, colocating the code paths for cached and uncached reads lets us implement both concisely, along with making it easy to fall back to the uncached mechanism if anything unexpected happens while reading the cache.

Writing caches, on the other hand, is a relatively simple action and effectively just consists of dumping the full verification info state to disk after the verification. For this reason, we add a separate Write Caches stage to the end of the verification runner that iterates over all modules and writes out both the read function dependencies and the verification info.

Figure 4.6 provides a graphical overview of the updated verification runner pipeline with caching.

4.6 Scripting Infrastructure

To this point, the verification framework has largely been concerned with manual usage, specific the scenario where a user invokes the client, packaged e.g. as a command-line tool, and then views pretty-printed, human-readable output. This works well for a number of use cases, specifically when prototyping or working on smaller projects. As with other compiler-related tools, e.g. analyzers, linters or optimizers, key benefits arise from the ability to automate them effectively. This includes programmability, but also integrability into build tools, package management, *Integrated Development Environments (IDEs)* or *Continuous Integration (CI)* pipelines.

4.6.1 JSON

To address this, the framework provides the underlying infrastructure for making verifications machine-readable by exposing APIs for converting the verification results to *JavaScript Object Notation (JSON)*, a ubiquitous format for semi-structured data. At the highest level, this means translating `VState` values to a JSON representation that is structured as follows:

```
{
  "<module 1>": {
    "<function 1>": "<info 1>",
    "<function 2>": "<info 2>",
    ...
  },
  "<module 2>": {
    ...
  },
  ...
}
```

This format is kept simple and includes only the computed verification info, not the programs themselves or other metadata from `VProgState`. This is an intentional design decision to hide implementation details and to be consistent with the pretty-printed output. It should also be noted that the JSON representation of the infos does not necessarily have to be a string, instead we let the client choose a suitable representation, which could also e.g. be an array or an object. This is reflected in the API, as the primary function for converting a `VState` to a JSON value takes a client-provided conversion from the verification-specific info to JSON, along with the `VState` itself:

```
vStateToJValue :: (a -> JValue) -> VState p a -> JValue
```

Here, JSON values are represented using the `JValue` type from the `json` package [Obe16b], which describes an abstract JSON element, i.e. an object, array, string, number, boolean or null value. For convenience, we also provide a slightly higher-level API that does not depend on any `json` package types, by leveraging the `ConvertJSON` type class that provides instances for all common JSON-convertible types, including `String`, `Int`, `[a]` and others:

```
vStateToJSON :: ConvertJSON b => (a -> b) -> VState p a -> String
```

In the simplest case, the client could use a function that pretty-prints the verification info to a `String`, or even pass the derived `show` function directly.

On the client-side there are a number of different ways to integrate the API, depending on how the verification framework is used. The perhaps most common use case is to provide a command-line option for outputting JSON and, following standard conventions, this amounts to adding a `--json` flag that, when set, outputs the final `VState` using one of the functions above. Any other output, including e.g. log messages, should be kept out of the standard output stream to avoid interfering with the JSON output, as this is specifically intended to be pipeable to a JSON file or other applications[10].

4.6.2 Automated Testing

One central application of the JSON API provided by the framework is testing, specifically checking whether verification outputs match a set of expected outputs. Previously, verification tools such as the non-failure verifier have used their own scripts to automate checking the tool's output against an expected output, this approach is not ideal, however: First, there is a fair bit of duplication when

[10]A simple way of doing so is by routing all other output to the standard error stream instead.

copying these scripts across different tools and, secondly, they depend on the precise output by the tool, making them brittle as small changes in the format will fail these tests.

For our framework, we provide a new command-line tool named `test-verification` that addresses these issues by providing a standardized way of writing tests for verifications in JSON. As input the tool takes the executable of a verification to test, along with the folder containing the expected JSON results for all modules to be verified and tested:

```
test-verification <executable> <results dir>
```

The implicit contract for using this tool is that the verification executable supports the `--json` flag as described in the previous paragraph, outputting the described JSON schema, and that the results directory contains a `<module>.json` file for each module to be verified, also following the schema. The modules must furthermore be available on the `CURRYPATH`. With this, invoking `test-verification` will run the verification all of these modules and compare the actual output against the expected one. For convenience, we only check whether the expected result is a *subset* of the actual result, to allow for omitting uninteresting verification results, e.g. from dependencies like the **`Prelude`**. If there are any mismatches in the expected verification infos, the tool will display the mismatch between the expected and actual output. It should be noted that all comparisons uses value equality on the JSON objects, thus it does not matter if the keys are ordered differently as long as the results are semantically equal.

Chapter 5

Case Studies

While the implementation presented in chapter 4 provides a broad foundation for new verifications, applying it to concrete use cases and studying the process of doing so is just as important to get a comprehensive picture of the framework's strengths and weaknesses. In this chapter we will therefore present two case studies that apply the framework to practical scenarios: We first provide an in-depth overview of porting one of the existing verifications, namely the contract prover, in section 5.1 and then implement a new verification from scratch in section 5.2. The latter is a termination checker that extends the ideas of the termination analysis from CASS with an implementation of a more general termination criterion, the so-called size-change principle, taking full advantage of the high-level abstractions provided by the verification framework.

5.1 Porting Existing Verifications

Existing verifications like the contract prover and the non-failure verifier have inspired significant parts of the framework's architecture and implementation, consequently adapting one of these tools to the framework is the first step towards making sure that the framework serves its purpose of being a comprehensive abstraction well. While we have already discussed many small examples to illustrate various abstractions in the framework earlier, this case study will focus on the process of porting the contract prover to the framework, both in terms of approaches taken and observations made.

To allow for better comparability between the original implementations and the framework-based ports, we maintain the same command-line interface and, where possible, try to keep the output of the tool close to the original. To achieve this, we hook into the framework's unified abstractions for logging and pretty-printing, e.g. by adapting the tool's existing logging infrastructure and by adding custom pretty-printing implementations for our verification info type. Additionally, we keep the existing implementation around behind a `--legacy` flag, to make comparisons between the original and the port even more seamless, as only a single version of the tool has to be built. We, however, also make sure that the port does not depend on anything from the legacy module, making it easy to remove the now isolated legacy implementation.

Porting the verification itself to the framework is largely a refactoring task. After moving the current implementation to a legacy module, we stub out a framework-based verification, which will be called by default unless the `--legacy` flag is set, and begin adapting chunks of the legacy implementation to the framework's verification lifecycle. Specifically, this entails deciding which parts of the existing implementation constitute preprocessing, initialization or iteration logic and then extracting the relevant parts. Holistically speaking, we thus employ a combination of top-down and bottom-up refactoring: By starting out with a skeletal implementation of a framework-based verification we go top-down and by gradually moving over parts of the verification we go bottom-up.

In the following we will take a closer look at how the contract prover's architecture maps to the framework and how it can be ported. Additionally, we will discuss some lessons learned, along with a review of advantages and shortcomings of the framework when applied to an existing tool in practice.

5.1.1 Contract Prover

Our first case study involves porting the contract prover [Han17a] to the verification framework. As sketched out in section 2.2.2, the contract prover is a relatively straightforward verification that does not perform any inference on its own and instead relies on explicitly defined contracts, i.e. pre- and postconditions, that are translated to SMT assertions and then checked by the Z3 solver. If the proof fails, the tool will insert dynamic checks to enforce any unproven contracts at run time. The simplicity of the contract prover makes it a good first candidate for porting a verification to the framework and, in fact, much of the framework has been developed in parallel with this port. Additionally, the insertion of dynamic checks has proven useful in evaluating the framework's ability to mutate the FlatCurry program during the verification.

The first step is to define a suitable verification info type, i.e. what we wish to compute for each function. Formalizing the example given in the description of the Initialize Verification Info stage from section 2.2.2, the contract prover's main job is to extract and verify pre- and postconditions found in the specified modules, thus the info type should include these conditions. For the results, we also need a flag that records whether the verification of a specific condition has succeeded. We begin by declaring a type named `Cond` that encapsulates a single pre- or postcondition:

```
data Cond = Cond
  { cName     :: String -- The pre/postcondition's name
  , cVerified :: Bool   -- Whether the condition could be verified
  }
  deriving (Read, Show, Eq)
```

Then we compose the verification info type `ContractInfo` from a list of pre- and postconditions:

```
data ContractInfo = ContractInfo
  { ciPreConds  :: [Cond] -- The function's preconditions
  , ciPostConds :: [Cond] -- The function's postconditions
  }
  deriving (Read, Show, Eq)
```

Recall that we need to make these types `Read` and `Show`, so the framework can serialize and deserialize these types for caching purposes and `Eq` for diffing purposes, hence the `deriving` clauses.

Using these types we can sketch out the central architecture of the contract prover in terms of the framework: The contract prover operates in terms of type-annotated FlatCurry with the ability to export either type-annotated or untyped FlatCurry, thus we also use type-annotated parameterization of the framework. The main implementation of the verification effectively just delegates to three methods, named `preprocessProg`, `initFuncInfo` and `analyzeFunc`, with `Options` encapsulating contract prover-specific command line options. At the highest level, we thus get the following functions:

```
contractProver :: Options -> TVerification ContractInfo
contractProver opts = emptyVerification
  { vPreprocess = preprocessProg
  , vInit       = initFuncInfo
  , vAnalyze    = analyzeFunc opts
  }

preprocessProg ::            VTProgEnv ContractInfo -> VM VTProgUpdate
initFuncInfo   ::            VTFuncEnv ContractInfo -> VM (Maybe ContractInfo)
analyzeFunc    :: Options -> VTFuncEnv ContractInfo -> VM (VTFuncUpdate ContractInfo)
```

The preprocessProg implementation, corresponding to the example given earlier in section 4.4.2, checks that all contracts a given program are used correctly, i.e. that for every function of the form f'pre or f'post an associated function f exists in the same module with a matching type. The initFuncInfo function, which operates on the function-level, when given a function f finds associated contracts like f'pre or f'post and returns a ContractInfo. Note that initFuncInfo only considers the main functions themselves as starting points and skips the contracts themselves by returning Nothing during their traversal. Finally, the analyzeFunc implementation includes the heart of the contract prover, i.e. attempts to both prove any calls with preconditions in the current function and prove the associated postcondition of the function itself. Any unsuccessful proofs are turned into dynamic checks using checkPreCond and checkPostCond, following the scheme detailed in the Iterate Verification Info example from section 4.4.2.

Internally, the implementation of analyzeFunc encapsulates a fair bit of complexity and thus uses its own monad. This largely still corresponds to the TransStateM monad from the original implementation of the contract prover, however now wraps the VM monad to support global, framework-level error throwing and separates readable and writable state more strictly:

```
type TransM = StateT TransState (ReaderT TransEnv (WriterT [TAFuncDecl] VM))
```

Here, the TransState and TransEnv types include internal state used by the assertion collecting mechanism, which, at a high level, corresponds to the ideas described in section 2.2.2, along with a mutable version of the current function, for adding dynamic checks. The [TAFuncDecl] writer collects new functions to be added to the program, also to support dynamic checks, all of which will be included in the VTFuncUpdate returned by analyzeFunc.

The purpose of having internal state and a separate monad in addition to the existing ContractInfo and verification infrastructure is twofold: First, we can hide implementation details about the assertion-collecting that would otherwise be included in the final output of the verification. Secondly, we get a cleaner separation of concerns between the contract prover and the verification framework by handling function-level traversals in the framework and expression-level traversals in the contract prover. More generally, this is a tradeoff in the framework's design: We intentionally require verification clients to perform expression-level logic, giving up on potential abstraction opportunities over assertion collecting, while simultaneously keeping the framework general enough to handle non-SMT-related verifications too.

It should also be noted that the implementation of the contract prover no longer includes a simplification mechanism, as this is handled by the Simplify stage at the framework-level. Specifically the desugaring of choices to FlatCurry Or nodes and binary operators to SMT primitives has largely been moved to the framework, only the mapping of function names to primitive SMT operation names has to be passed via the VOptions to the framework's entry point. As outlined in section 4.4.2, we use the flatcurry-smt package to provide these mappings by using its unaryPrimOps and binaryPrimOps.

Examples

One of the simplest examples of the contract prover is the coin function, with a postcondition asserting that the result must be positive:

```
coin'post :: Int -> Bool
coin'post r = r > 0

coin :: Int
coin = 1 ? 2
```

Both the existing and the framework-based contract prover successfully verify this postcondition:

```
Coin:
  coin ->   VERIFIED: coin'post
ALL CONTRACTS VERIFIED!
```

Another example is the `last` function, which retrieves the last element of a list. This example includes a precondition on the argument that asserts that the list is not empty, thereby also introducing a dependency on the `Prelude`, due to the `not` and `null` functions:

```
last'pre :: [a] -> Bool
last'pre xs = not (null xs)

last :: [a] -> a
last [x]      = x
last (_:x:xs) = last (x:xs)
```

Likewise, both the existing and the framework-based version of tool handle the verification successfully:

```
Last:
  last  ->   VERIFIED: last(last)
Prelude:
  ...
ALL CONTRACTS VERIFIED!
```

5.1.2 Observations and Lessons Learned

A central observation from porting the contract prover is that while the abstractions of the framework provide a convenient interface for expressing fixed-point verifications, there are still domain-specifics that either cannot be abstracted over well at the framework-level or are sufficiently complex enough to warrant a custom abstraction in the client itself. The contract prover, for example, implements a custom `TransM` monad to abstract over the state involved in the translation to SMT and effectively wraps the framework's verification environment and output in a higher-level interface that is suitable for the specific verification. Additionally, much of the contract prover's implementation consists of specific transformations that cannot be shared with other verifications, therefore the framework-based verification is only slightly shorter than the original tool. There are, however, a number of nice advantages about using the framework: For example, the abstraction over the handling of verification values makes it easier to integrate features such as caching or structured JSON output centrally and provides a central and optimized facility for state management around FlatCurry programs. Finally, the framework-based contract prover will also be able to take advantage of generic optimizations of the fixed-point iteration at the framework-level, such as improved dependency tracking, which can now be implemented generically across verification tools.

5.2 A Termination Checker Based on the Size-Change Principle

Our second case study consists of implementing a non-trivial verification from scratch, specifically a termination checker that combines ideas from the termination analysis from CASS with a powerful termination criterion, the size-change principle. A key goal of the framework is making it easy to implement new verifications and in doing so we can identify strengths and weaknesses of our architecture when applied to a novel problem. At a high-level, our termination checker will attempt to prove termination for each function of the given set of modules, then output for which ones the proof succeeds and for which it does not. In the following sections, we will outline the problem, by clarifying what termination checking means, introduce termination criteria leading up to the size-change principle and finally present our approach and implementation.

5.2.1 Termination Checking

Classically, *termination checking* refers to the decision problem of deciding whether a program, or, without loss of generality, a function, terminates on all inputs. When phrased as a decision problem like this, it is also known as the *halting problem*, which is famously undecidable [Dav58][1]. Writing a program that for *any* program decides whether it terminates or not is thus impossible. To get around this, we approach this problem conservatively, by trying to prove termination in as many cases as possible and falling back to the default answer, stating that the termination behavior is unknown. To distinguish these cases, we call them *terminating* and *possibly looping*.

In the context of functional languages like Curry or Haskell it is often more useful to talk about the behavior of a function rather than a program in the imperative sense, given that the notion of a "program" is imprecise: Many Curry programs are libraries and do not have a `main` function, yet it would still be useful to make statements about such programs. Unlike in strict languages, however, where functions can effectively be considered procedures or subprograms, we need to clarify what termination of a function means in a lazy language. In a lazy language, the termination behavior of a function or value depends on how it is used, thus an expression like `head (repeat 1)`, would terminate when evaluated to NF whereas `repeat 1` would not, as the latter would try forcing the entire infinite list. We therefore choose the definition of *NF-termination* [Mit04], under which we consider a function `f` to be terminating iff `f x1 ... xn` reduces to NF for any arguments `x1, ..., xn` in NF. A corollary of this is that values, when interpreted as 0-ary functions, terminate under our definition iff they have an NF. Note that infinite data structures such as the list constructed by `repeat 1` do *not* have an NF as they can always be reduced further, thus our definition intentionally excludes making any statements on the termination behavior of expressions like `head (repeat 1)`. Function expressions, on the other hand, may have an NF if their body can be reduced fully, thus we can talk about the termination behavior of expressions involving functions, such as `map (*2) [1, 2, 3]`.

For Curry specifically, we also need to clarify the difference between non-termination and divergence. We consider diverging functions, such as `failed`, to be terminating too if all branches of the computation eventually result in failure. This lets us make some simplifying assumptions, e.g. that all primitive and external `Prelude` functions are terminating.

5.2.2 Decreasing Recursion

To make statements about a function's termination behavior, it is useful to consider the cases under which a function would *not* terminate and under our definition there is only one such case, namely when a function eventually loops on some input. Given that the only way of introducing loops in Curry is via recursion, it is sufficient to show that a function does not recurse forever, i.e. that all call paths must eventually end and thus that the function in question terminates.

Two key observations provide us a simple, but already remarkably general criterion for this: First, note that when a function calls itself, the structure of its parameters and the arguments of the recursive

[1]Interestingly, the origins of the halting problem seem to be less clear than often assumed. Despite often being credited to Alan Turing's landmark paper [Tur36], [Luc21] finds that Turing's decision problems about computable numbers were distinct from the halting problem, whose first reference appears to be [Dav58], hence that is the citation we use.

call have the same arity and types. Secondly, functions commonly destructure their parameters and then use these subterms in their recursive calls. If we can infer a relationship between the parameter and the recursive call argument at *any* parameter position such that the recursive call argument is a subterm of the parameter, we get a strictly monotonically decreasing sequence of argument values over the total call sequence. Such a sequence must end if the values in question are finite, thus the function must terminate. To illustrate this, consider the function that computes the length of a list:

```
length :: [a] -> Int
length []     = 0
length (_:xs) = 1 + length xs
```

This function has a single recursive call for which it passes `xs`, a subterm of the destructured parameter `(_:xs)`. The criterion introduced above tells us that for any finite input list, the `length` function will terminate, as every recursive call will strictly decrease the size of the list. And indeed, passing any list `[x1, x2, ..., xn]` to the function results in the finite call sequence

`length [x1, x2, ..., xn]` → `length [x2, ..., xn]` → ... → `length [xn]` → `length []`

The same argument can be applied to a number of other list traversals, including higher-order functions like `map` and `foldr` from the **`Prelude`**. In fact, the criterion already generalizes to arbitrary algebraic data types, allowing it to handle any divide-and-conquer-style traversal where the function directly recurses on a part of its input. It is worth emphasizing that it does not matter whether other arguments increase as long as there is one argument which decreases. Consider, for example, the `reverse` function, which can be implemented efficiently using the accumulator technique by defining it as `reverse' []` where `reverse'` is implemented as follows:

```
reverse' :: [a] -> [a] -> [a]
reverse' acc []     = acc
reverse' acc (x:xs) = reverse' (x:acc) xs
```

Even though the accumulator argument `acc` gets larger in every step, the second argument decreases strictly, therefore `reverse'` terminates per the criterion. One limitation is that the criterion currently only handles functions that call themselves, thus `reverse` could not be handled yet. There is, however, a relatively simple extension to the idea that handles this: A function calling other functions terminates iff the criterion applies to its recursive calls and the other functions it depends on can be proven to terminate[2]. This extended criterion corresponds to the implementation of termination analysis in CASS and can successfully prove termination of larger programs.

A central limitation remains, however: Termination can only be proven as long as there is no mutual recursion, i.e. the function call graph does not contain strongly connected components of size greater than 1. This is a bit unfortunate in practice as a number of essential functions from the **`Prelude`** rely on this. Take, for example, the `take` function, which returns the first `n` elements of a list `l`:

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

The `l` parameter is destructured to `(x:xs)`, from which `xs` is used in the recursive call to `take`. Even though every recursive call to `take` strictly decreases the second argument, the current criterion is not able to see through this recursion because the intermediate call to `takep` appears to keep the argument

[2]Attentive readers may already notice how this maps to the verification framework's fixed-point iteration

the same, thus not decreasing it strictly. The dependency check also fails as both functions have a dependency on each other, thus we cannot check them in topological order and use the termination behavior of one to infer the other's. In other words, we need a stronger criterion.

5.2.3 Size-Change Principle

The *size change-principle* provides such a criterion by generalizing the previous approach to allow for arbitrarily sized call cycles across different functions. At a high level, the size-change principle states the following [LJBA01]:

> If every infinite computation would give rise to an infinitely decreasing value sequence (according to the size-change graphs), then no infinite computation is possible.

Recall that the only way to create an infinite computation in Curry is via recursion, therefore we can rephrase this as stating that a function terminates if there is an argument that decreases strictly for every recursive call, regardless of which functions are called in between. It should be noted that the notion of "decreasing" in this statement applies to any well-founded[3] partial order on values, thus generalizing the idea of subterm relationships discussed in the previous paragraph to allow for other orderings. We will discuss specific examples later.

To understand both the formal theorem describing the size-change principle in detail and the algorithm implementing it, we will introduce some definitions that operate on the FlatCurry syntax introduced in section 2.1.5. These closely follow those provided by Lee et al. in [LJBA01], but are adapted for simplicity and the use of FlatCurry where needed.

Let P be a FlatCurry program and $f(x_1, ..., x_n) = e$ be a function defined in P. For notational convenience, we define $f^{(i)} = x_i$ to be the i-th parameter of f and $Param(f) = \{f^{(1)}, ..., f^{(n)}\}$ to be the set of parameters of f. We assume each function call expression $g(y_1, ..., y_m)$ occurring in e has a label c that is unique throughout the program P and denote each call by $c : f \to g$. Let C_P be the set of all calls in P. We will refer to both finite and infinite sequences $c_1 c_2 ...$ of calls in C_P as *call sequences*.

For every call $c : f \to g$ in P, a *size-change graph* $G : f \to g$ is a bipartite graph of the form

$$G = (Param(f), Param(g), E)$$
$$E \subseteq Param(f) \times \{\downarrow, \overline{\downarrow}\} \times Param(g)$$

where for each edge $f^{(i)} \xrightarrow{w} g^{(j)} \in E$ the pair $(f^{(i)}, g^{(j)})$ uniquely determines the size-change relationship w, which we denote by either $\downarrow$ iff $g^{(j)} < f^{(i)}$ or $\overline{\downarrow}$ iff $g^{(j)} = f^{(i)}$ for some well-founded partial order $<$ on FlatCurry values. There are different orderings we can choose here, including the previously used subterm relationship $<_{term}$, under which $x <_{term} y$ iff x is a subterm of y and $x \neq y$, or the absolute value ordering $<_{abs}$ where $x <_{abs} y$ iff $|x| < |y|$ for $x, y \in \mathbb{Z}$. For natural numbers, the natural order $<$ is already well-founded, thus could also be used, though $<_{abs}$ tends to be more convenient in practice as it covers all integers and thus does not require us to prove non-negativity separately.

To illustrate the notion of size-change graphs, consider the `take` function defined earlier. We can define size-change graphs G_1 : `take` $\to$ `takep` and G_2 : `takep` $\to$ `take` for the two respective calls with respect to $<_{term}$ as shown in figure 5.1: In the call from `take` to `takep`, the parameters stay equal, thus we can use $\overline{\downarrow}$ edges, whereas in the call from `takep` to `take`, the `(x:xs)` argument is shrunk to the subterm `xs`, therefore we can add a $\downarrow$ edge to signify that the value decreases strictly. Notice that size-change graphs are not determined uniquely by the FlatCurry function declaration: Any subset of

[3]A relation $<$ is well-founded if there are no infinitely decreasing sequences of the form $x_1 > x_2 > ...$ or, equivalently, if every non-empty subset of values has a least element with respect to $<$. For us, particularly the first characterization is of interest, as our termination criterion relies on the well-foundedness of the ordering relation.

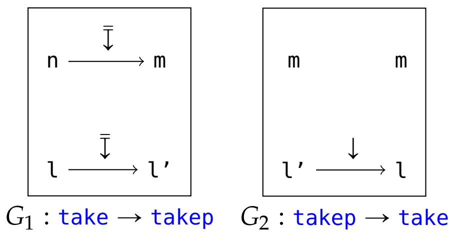


Figure 5.1. Size-change graphs for take and takep w.r.t subterm ordering

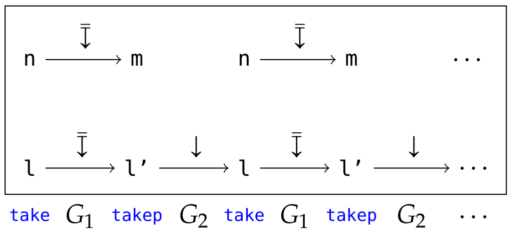


Figure 5.2. Infinite multipath $G_1 G_2 G_1 G_2 ...$ for take

the graph's edges still satisfies the definition of a size-change graph, including the trivial graph with no edges. This is expected as the termination checker is always able to make a weaker statement, or no statement at all, about the function's behavior.

Recall that a size-change graph only provides information about a single call. Since the size-change principle takes not just direct recursive calls into account, we need to consider multiple size-change graphs in sequence to analyze the termination behavior of a function. This leads us to the notion of a *multipath*, which is a finite or infinite sequence of size-change graphs $G_1 G_2 G_3 ....$ Expanding on the previous example, figure 5.2 shows an infinite multipath, illustrating the mutually recursive behavior of the take function. For a multipath we also define the notion of a *thread* as a, possibly infinite, sequence of connected edges $th = f_1^{(i_1)} \xrightarrow{w_1} f_2^{(i_2)} \xrightarrow{w_2} ...$ and call a thread *infinitely descending* iff there are infinitely many strictly descending edges, i.e. edges with the $w_j = \downarrow$.

Let $\mathcal{G}_P = \{G_c \mid c \text{ is a call in } P\}$ where each G_c is a[4] size-change graph for c and cs be a call sequence. We define $\mathcal{M}^{\mathcal{G}_P}(cs) = G_{c_1} G_{c_2} ...$ to be the $\mathcal{G}_P$-multipath for cs. The size-change termination principle can now be formulated as follows[5]:

> A program P is size-change terminating for a given choice of $\mathcal{G}_P$ iff for every well-formed infinite call sequence $cs = c_1 c_2 ... \in C_P^{\omega}$ there exists an infinitely descending thread th in $\mathcal{M}^{\mathcal{G}_P}(cs)$.

In the example of the take function, take → takep → take → ..., as shown in figure 5.2, is the only well-formed infinite call sequence, and it contains the infinitely descending thread $\mathtt{l} \xrightarrow{\bar{\downarrow}} \mathtt{l'} \xrightarrow{\downarrow} \mathtt{l'} \xrightarrow{\bar{\downarrow}} \mathtt{l}...$, hence the take function terminates. It should be emphasized that the statement only requires the

[4]As mentioned before, this set depends on the method for computing each G_c. Per [LJBA01] the most precise size-change graphs are uncomputable in the general case, but we can still approximate this set using techniques described later. For now, we will assume that we have a way of computing such an approximation.

[5]Side note: The original paper, i.e. [LJBA01], additionally defines the set of well-formed infinite call sequences as $FLOW^{\omega}$ and the subset of call sequences whose multipath contains an infinitely descending thread as $DESC^{\omega}$. The size-change termination principle can then also be characterized as stating that a program terminates iff $FLOW^{\omega} = DESC^{\omega}$. Since we leave the proof to the original paper, we consider this definition to be unnecessarily technical, therefore we choose to omit these definitions.

existence of a single infinitely descending thread per infinite call sequence, i.e. per cycle in the function call graph. In practice this means that only a single argument has to shrink from a function to its recursive call, possibly across multiple functions. The notion of threads also abstracts away from parameter order, allowing arguments to be swapped and rearranged freely, as long as the computed size-change graphs contain a path of smaller-or-equals relationships for the arguments in question where at least one edge is a "strictly smaller" relationship.

In addition to the theorem itself, Lee et al. describe a graph-based characterization in [LJBA01] that phrases the size-change termination principle in terms of a transitive closure over size-change graphs using a notion of graph composition. Like the termination principle, this characterization makes no assumptions about how the size-change graphs themselves are computed and, when interpreted algorithmically, only expresses the abstract search for an infinitely descending thread among the already computed set of graphs $\mathcal{G}_P$.

Let $G_1 : f \to g$ and $G_2 : g \to h$ be size-change graphs with edge sets E_1 and E_2, respectively. We define $G_1; G_2$ to be the *composition* of these graphs with

$$\begin{aligned} G_1; G_2 &= (Param(f), Param(h), E) \\ E &= E_{\downarrow} \cup E_{\overline{\downarrow}} \\ E_{\downarrow} &= \{x \xrightarrow{\downarrow} z \mid \exists y \in Param(g), w \in \{\downarrow, \overline{\downarrow}\} : x \xrightarrow{\downarrow} y \xrightarrow{w} z \text{ or } x \xrightarrow{w} y \xrightarrow{\downarrow} z\} \\ E_{\overline{\downarrow}} &= \{x \xrightarrow{\overline{\downarrow}} z \mid \exists y \in Param(y) : x \xrightarrow{\overline{\downarrow}} y \xrightarrow{\overline{\downarrow}} z \text{ and } x \xrightarrow{\downarrow} z \notin E_{\downarrow}\} \end{aligned}$$

where $x \xrightarrow{w} y \xrightarrow{w'} z$ is a shorthand for $x \xrightarrow{w} y \in E_1$ and $y \xrightarrow{w'} z \in E_2$. Per [LJBA01] graph composition is associative, as expected for a composition operator, thus lets us omit parentheses. We define the transitive closure $\mathcal{S}_P$ of size-change graphs over all well-formed finite call sequences as follows:

$$\mathcal{S}_P = \{G_{c_1}; G_{c_2}; ...; G_{c_n} \mid cs = c_1 c_2 ... c_n \in C_P^* \text{ well-formed and } c_1 : f \to g\}$$

This lets us characterize the size-change termination principle as follows:

> A program P is size-change terminating iff for every $G : f \to f$ in $\mathcal{S}_P$ with $G = G; G$ the graph G has an edge of the form $x \xrightarrow{\downarrow} x$.

Comparing this characterization with the previous one reveals a close similarity: Infinite call sequences are characterized as cyclic graphs, i.e. graphs of the form $G : f \to f$, that are idempotent under composition, threads are represented by edges after composition and infinite descent can be detected by checking the size relationship for strict decreasing, i.e. $\downarrow$.

5.2.4 Data Model

Before diving into the implementation of a termination checker based on the size-change principle, we need to clarify what we wish to compute, i.e. the type of the verification values that we want to assign each function via the framework. A key constraint is that the verification framework requires clients to bundle all state that they wish to persist across the entire verification in the verification values, thus we include both the size-change graphs and the info about the final termination behavior, the "actual" output, in the verification info type, which looks as follows:

```
data TerminationInfo = TerminationInfo
  { tiTerminating :: Terminating
  , tiGraphs      :: Set Graph
  }
  deriving (Read, Show, Eq)
```

Here tiTerminating represents the termination behavior of the function, which we model as follows:

```
data Terminating = Terminating     [String]
                 | PossiblyLooping [String]
                 | Unknown         [String]
  deriving (Read, Show, Eq)
```

We distinguish between three cases, Terminating to signify that the function terminates, PossiblyLooping to express that a function may not terminate and Unknown to distinguish unverified functions from verified ones. Each of these cases is equipped with a list of strings containing a human-readable list of reasons why the termination checker made this classification. The distinction between terminating and possibly looping functions corresponds to the idea presented in section 5.2.1, namely that we intend to prove termination with certainty, i.e. without false negatives, but fall back to a case expressing uncertain behavior, under which a function may or may not loop.

The set tiGraphs, on the other hand, represents the transitive closure over size-change graphs in the associated function and is roughly analogous to the $\mathcal{S}_P$ set from the previous paragraph. It should be noted, however, that while $\mathcal{S}_P$, like the other definitions from the previous section, is based on a program P as the top-level syntactic element, our verification framework operates in terms of functions, therefore we compute the graphs starting from a single, namely the verified, function. Since we do, however, successively include all callees transitively and a function can always be wrapped in a program, the previous section's ideas still largely apply verbatim. The Graph type that this field uses to represent size-change graphs is largely modeled after the mathematical description:

```
data Graph = Graph
  { gCall  :: Call
  , gEdges :: Map Edge SizeChange
  }
  deriving (Read, Show, Eq)
```

Each graph represents a single Call, stored in the gCall field, which is a pair of a source function (the caller) and a destination function (the callee), both represented as qualified names:

```
data Call = Call
  { cSrcFunc :: QName
  , cDstFunc :: QName
  }
  deriving (Read, Show, Eq, Ord)
```

The edges are stored in gEdges and capture known size-change relationships. Each Edge represents a relationship between a pair of a source function parameter and a destination function parameter index, both zero-based:

```
data Edge = Edge
  { eSrcParam :: Int
  , eDstParam :: Int
  }
  deriving (Read, Show, Eq, Ord)
```

These are mapped to SizeChange values, representing the $\downarrow$ and $\overset{=}{\downarrow}$ arrows, respectively:

```
data SizeChange = SmallerSize | EqualSize
  deriving (Read, Show, Eq, Ord)
```

Here, the `Ord` instance also makes sure that we can compare the size-changes using the standard comparison operators, a property that we will use later during the implementation of composition.

5.2.5 Implementation

The implementation of the termination checker is, at the highest level, a type-annotated verification that computes a `TerminationInfo` for each function. The choice to use type-annotated FlatCurry has practical reasons related to the SMT translation, which we will discuss later. In code, the `Verification` is defined as follows:

```
checkTermination :: Options -> TVerification TerminationInfo
checkTermination opts = emptyVerification
  { vInit    = initTerminationInfo opts
  , vAnalyze = analyzeFuncTermination
  }
```

Our approach is to split up the problem of termination checking into two phases: During the initialization, i.e. in `initTerminationInfo`, we compute the size-change graphs for each function based on different strategies. During the fixed-point iteration these graphs are repeatedly composed and checked for infinitely descending threads to determine the termination behavior. This is implemented in `analyzeFuncTermination`, which computes both the next level of graph compositions and an updated `TerminationInfo`. In the following we will describe these two phases in detail.

Initialization Phase: Computing Size-Change Graphs

During the initialization phase, we compute the initial `TerminationInfo` for each function. The interesting part here is the computation of the initial set of size-change graphs, for the termination behavior itself we just use the `Unknown` case for now. Since the verification framework only performs a single pass over all tracked functions during the Initialize Verification Info stage, we limit ourselves to local information here, i.e. we compute the size-change graphs for all *direct* calls in each function. This can be done in isolation for each function as we only need to consider the callees which we can obtain by syntactically traversing the expression for the function body. Since size-change graphs are all about the size relationships we can infer about each call's arguments compared to the function arguments, we still need to consider the surrounding context and cannot just look at the calls themselves. For this, we employ two main strategies, which we run separately over the function declaration:

1. A purely syntactic strategy that traverses the function body expression and keeps track of subterm relationships between variables and the function arguments using a mechanism we term *abstract values* to compute size-change graphs based on these subterm relationships.

2. An SMT-based strategy that translates the entire function to an SMT-LIB program, containing size-change assertions for each call argument, hands it to the Z3 solver and then constructs the size-change graphs by parsing the output.

Since both strategies are implemented in isolation, we will discuss these separately before going into detail on how the results are combined.

The Subterm-Based Strategy The subterm-based strategy is a simple, yet powerful strategy that infers syntactic size relationships between the function arguments and the arguments of any calls occurring in the function body. The basic idea is based on the termination analysis from CASS, which passes down a mapping of function arguments to all variables that have been discovered to be smaller and accepts a program as terminating iff at least one argument shrinks in a direct recursive call[6]. Although the termination analysis from CASS correctly identifies pattern arguments in `case` expression branches to be smaller, however fails to keep track of size-change relationships where expressions are not just destructured, but also constructed, i.e. made bigger. To give a practical example, consider the following function implementing the "less or equals" operator on Peano numbers:

```
leq :: Peano -> Peano -> Bool
leq Zero      _         = True
leq (Succ _)  Zero      = False
leq (Succ p1) (Succ p2) = leq p1 p2
```

where `data Peano = Zero | Succ Peano`. The termination analysis of CASS is able to prove this function as being terminating, as even after the desugaring of function rules to `case` expressions in FlatCurry, the function only destructures its arguments. If we, however, rewrite this function very slightly to match on a tuple instead[7], CASS is no longer able to prove that this function terminates:

```
leq :: Peano -> Peano -> Bool
leq p1 p2 = case (p1, p2) of
  (Zero    , _       ) -> True
  (Succ _  , Zero    ) -> False
  (Succ p1', Succ p2') -> leq p1' p2'
```

The issue is that the discriminating expression, `(p1, p2)`, constructs, with respect to the subterm ordering, a term that is larger than both `p1` and `p2`, for which the CASS analysis is unable to infer a size relationship to the function arguments. Since the discriminating expression is thus effectively an opaque value, no size relationships can be inferred about the pattern variables can be inferred either, even though they are semantically equivalent to the function arguments.

To solve this, we introduce the concept of *abstract values*, which are a tree representation of a term where the leaves capture the size relationship to a function argument and the branches represent constructors. This makes it possible to keep track of size relationships across arbitrary levels of constructing (i.e. expressions) and destructuring (i.e. pattern matchings). We define an abstract value as follows:

```
data AbstractValue = AbstractCons QName [AbstractValue]
                   | AbstractArgRelated SizeChange Int
                   | AbstractOther
  deriving (Show, Eq)
```

Here, `AbstractCons` represents a constructor call, i.e. a branch with abstract values representing its arguments, whereas `AbstractArgRelated` is a leaf that represents a value with a known given size relationship to a function argument with a specific index and `AbstractOther` is a leaf representing a value without known size relationships. To illustrate this, consider the `leq` function we

[6]The limitation of only handling direct recursive calls does not apply to our new termination checker due to the iteration phase, which we will discuss later, here we will focus solely on the inference of the size change relationships.

[7]This is, in fact, a common trick to get Haskell-style sequential pattern matching in Curry even when the patterns are not orthogonal. In this case it does not matter, as the rules are orthogonal, but in other cases it does, especially when attempting to use wildcard pattern as a "fallback" branch.

discussed above: The `p1` and `p2` arguments would be represented as `AbstractArgRelated EqualSize 0` and `AbstractArgRelated EqualSize 1`, respectively. For the term `(p1, p2)` aka. `(,) p1 p2` we would get the following abstract value:

```
AbstractCons ("Prelude", "(,)")
  [ AbstractArgRelated EqualSize 0
  , AbstractArgRelated EqualSize 1
  ]
```

When pattern-matching on the tuple we "peel" off the outer `AbstractCons` and when pattern-matching even further on the `EqualSize` (or `SmallerSize`) value, all matched pattern variables turn into a `SmallerSize` abstract value with respect to the same function argument index. Since the strategy employed by CASS's termination analysis is effectively equivalent to abstract values without the `AbstractCons` constructor, our approach can be viewed as a direct generalization.

The actual algorithm of computing the size-change graphs from the function body is relatively straightforward: The idea is that we pass down a map from variables to abstract values during our traversal of the function body expression, which is updated whenever new values are bound, e.g. via a `let` expression or a `case` branch. In the implementation we first define an environment that stores both the parent environment from `initTerminationInfo` in `sgFuncEnv` and a map of variables to their abstract values in `sgVars`. Since FlatCurry already provides a unique index for each variable, we just use its `VarIndex` type[8], thus the whole type can be defined as follows:

```
data SGEnv = SGEnv
  { sgFuncEnv :: VTFuncEnv TerminationInfo
  , sgVars    :: [(VarIndex, AbstractValue)]
  }
```

For convenience, we wrap it into a reader monad named `SGM`[9], so we can write the traversal monadically, passing the environment as a parameter would, however, be semantically equivalent:

```
type SGM = Reader SGEnv
```

We can then implement a `graphsForFunc` function that computes size-change graphs purely from the function declaration itself, without any mutable state, I/O or other effects:

```
graphsForFunc :: TAFuncDecl -> SGM [Graph]
```

Internally, we first assign all function argument variables the abstract value `AbstractArgRelated EqualSize i` in the environment, where `i` is the argument index. Then, we traverse the expression recursively. When traversing the body of a `let` expression, we compute the abstract values for all newly bound variables from the environment and then pass the updated environment down the traversal. The conversion of a FlatCurry expression to an abstract value, as sketched out earlier, is relatively straightforward here and is implemented by the following function:

```
abstractValueForExpr :: SGEnv -> TAExpr -> AbstractValue
abstractValueForExpr env e = case e of
  AVar _ v                  -> fromMaybe AbstractOther . lookup v $ sgVars env
  AComb _ ConsCall (qn,_) es -> AbstractCons qn $ abstractValueForExpr env <$> es
  _                         -> AbstractOther
```

[8] `VarInt` is a synonym for `Int`.
[9] `SGM` is short for "subterm graph monad", for lack of a better name.

Notably, we look up variables in the environment, transform constructors and fall back to **AbstractOther** for anything else. Despite its simplicity, this strategy for computing size-change graphs handles a surprising number of cases, as constructing and destructuring terms containing data constructors already covers a large subset of Curry functions that operate on algebraic data types.

The SMT-Based Strategy One notable case where the subterm-based strategy falls short is on functions involving integer arithmetic. Although the subterm strategy often still proves useful, as only a single shrinking subterm-based argument relationship has to be proven, it cannot handle cases where *all* arguments are integer arithmetic, for example. Consider the following version[10] of the factorial function and how the only argument to the recursive call is an arithmetic expression:

```
fac :: Int -> Int
fac n = if n == 0 then 1
                  else n * fac (n - 1)
```

For the subterm-based strategy this is unfortunate as the `abstractValueForExpr` function will fall back to **AbstractOther** for this call argument, thus the termination checker cannot prove anything about this factorial function via the subterm strategy[11]. As mentioned earlier, there are other orderings that we can use infer a size relationship, specifically the absolute value ordering $<_{abs}$ that orders numbers by magnitude. Proving that any size relationship between numeric values is, however, a much harder task if we consider all the different ways in which an expression could grow or shrink via arithmetic compared to the purely syntactic construction and destructuring of constructor terms as described in the previous section. Fortunately, SMT solvers like Z3, as outlined in section 2.2.1, already implement a wide range of algorithms to deal with this problem of proving statements over integers, booleans and a number of other primitives, therefore our solution is to translate the function to an SMT-LIB program from whose output we can directly construct the size-change graphs by parsing it.

The SMT Translation The key idea behind our SMT-based size-change graph construction is to generate two SMT assertions for every pair of a function argument and call argument in the function to be verified, both of which are to be proved separately: In the first one we want to prove that the call argument is strictly smaller than the function argument with respect to $<_{abs}$ and for the second one that the call argument is equal to the function argument. In other words, we effectively try proving every possible combination of **Edge** and **SizeChange**. There is, however, a very important consideration that we need to make here: An SMT solver only proves *satisfiability* of the specified assertions, i.e. tries finding an example that satisfies them simultaneously, effectively like proving an existentially quantified statement. In our case, however, we want to show that a size change relationship *always* holds, as e.g. a recursive call argument that only shrinks under some circumstances does not prove us anything. Consider the example of the factorial function, as we try proving the size-relationship $<_{abs}$ of the recursive call argument in the **else** branch, thus already under the assumption that $n \neq 0$, using the following, simplified SMT program:

```
(declare-const n Int)
(assert (not (= n 0))) ; else branch
```

[10]This version is intentionally defined using a catch-all **else** branch, unlike the definition of `fac` in section 2.1.5, for the demonstrating edge cases under the naive SMT translation.

[11]It should be noted that this could, to a certain extent, be circumvented by just defining arithmetic operators in terms of Peano numbers, which are algebraic data types. Although theoretically elegant, they are cumbersome to work with in practice and perform much worse than native integers, hence they are only really useful in narrow cases, particularly when working with nondeterminism in the absence of other integer constraint solving techniques.

```
(assert (< (abs (- n 1)) (abs n)))
(check-sat)
(get-model)
```

The solver tells us that this assertion is satisfied by $n = 1$, but we are effectively just proving $\exists n \in \mathbb{Z}\backslash\{0\} : |n-1| < |n|$ while we actually want to prove $\forall n \in \mathbb{Z}\backslash\{0\} : |n-1| < |n|$. Fortunately, De Morgan's law tells us that the negation under the existential quantifier is equivalent to the negation over the universal quantifier, i.e. that $\exists n \in \mathbb{Z}\backslash\{0\} : \neg(|n-1| < |n|)$ is equivalent to $\neg\forall n \in \mathbb{Z}\backslash\{0\} : |n-1| < |n|$, hence asserting the negation and then flipping `unsat` and `sat` in the output lets us prove the universally quantified version of the statement. In other words, we would replace every size relationship assertion with its negation, in our case

```
(assert (not (< (abs (- n 1)) (abs n))))
```

Unfortunately, the statement is still satisfiable, which means that $\forall n \in \mathbb{Z}\backslash\{0\} : |n-1| < |n|$ could, in fact, *not* be proven. Naively this would mean that our termination checker could still not prove that `fac` terminates, which would, in fact, hold true here as `fac (-1)` would loop infinitely. We can, however, also assert the precondition `fac'pre n = n >= 0`, that traditionally would have been verified by the contract prover, in the translated SMT program:

```
(declare-const n Int)
(assert (>= n 0))      ; precondition
(assert (not (= n 0))) ; else branch
(assert (not (< (abs (- n 1)) (abs n))))
(check-sat)
```

Now the program is unsatisfiable, which means we have proved that the magnitude of the recursive call argument shrinks and thus that the function terminates.

The algorithm for translating an arbitrary FlatCurry function to such an SMT program is slightly more complex, but builds upon the same idea. At the highest-level, when given a FlatCurry function declaration of the form `f(`x_1`, ..., `x_n`) =` e, we first check whether an associated precondition of the form `f'pre(`x'_1`, ..., `x'_n`) =` e' exists, then generate the following SMT code:

```
; Arguments
(declare-const SMT(x_1) SMT(Type(x_1)))
...
(declare-const SMT(x_n) SMT(Type(x_n)))

; Preconditions (omitted if non-existent)
(assert SMT(e'[x'_n ↦ x_n]))

; Body
...
```

where for a FlatCurry expression x the notation $x[\overline{y_m \mapsto z_m}]$ denotes an updated version of the expression where all substitutions $y_i \mapsto z_i$ have been applied for $i \in \{1, ..., m\}$, $Type(x)$ denotes the type-annotation of a FlatCurry expression[12] x, $SMT(x)$ denotes the SMT translation of any given FlatCurry construct x. For the latter, we leverage the existing infrastructure for translating FlatCurry expressions, patterns and types to SMT, provided by the `Curry2SMT` module already used by the contract prover and the non-failure verifier.

[12]This requires type-annotated FlatCurry and is the primary reason why we use it over untyped FlatCurry for this verification.

For translating the function body we use a scheme that translates the FlatCurry expression directly to SMT code, emitting assertions for every call, specifically every combination of function argument and call argument. The output of the SMT solver, in our case Z3, can then be used to directly construct a size-change graph for each call. To handle assertions about different code paths and also to e.g. verify different size-change relations separately, we use scopes introduced by `(push)` and `(pop)`, thus leveraging the SMT assertion stack for tracking the known information about variables in each syntactic scope without affecting anything outside it.

Figure 5.3 shows the full translation scheme in the form of operational semantics using judgments of the form $e \Downarrow S$ to denote that a FlatCurry expression e is translated to an SMT program S. We use the $\wedge$ operator to concatenate SMT assertions[13], i.e. $S_1 \wedge S_2 = S_1 S_2$ for SMT programs S_1 and S_2. The rules can now be understood as follows:

- ▻ The first rule, (Prim), is effectively a fallback rule that translates all variables, literals and `free` declarations to the empty SMT program. We do not handle these as they do not contain any subexpressions, calls or are otherwise relevant to the SMT translation. Note that this only affects our top-level traversal of the function body expression, variables and literals are still translated to SMT terms as part of the SMT function, which we use to generate the actual assertions.
- ▻ The second rule, (Or), is also relatively simple: For a nondeterministic choice we simply traverse both branches of the choice and concatenate the corresponding SMT translations. Note that each of the translations get a `(push)`/`(pop)` scope so the individual assertions do not affect each other. This is a technique that we will make use of liberally during the rest of the translation.
- ▻ The third rule, (Cons), handles constructors, whose arguments we just traverse in order to handle any nested calls, adding a scope around each translated argument and concatenating the resulting SMT code.
- ▻ The fourth rule, (Fun), is perhaps the most interesting one, as it translates function call expressions. We first output a short message via `(echo)` containing the name of the called function, so our parser can construct the resulting size-change graph more easily without having to know about the precise ordering of the calls in the function. Then, we traverse the call arguments, appending each translation to the SMT output in a scope and finally add a scoped size-change assertion for each combination of function argument, call argument and size relation to allow each possible size relation to be checked separately.
- ▻ The fifth rule, (Let), handles `let` expressions. Here, we translate each variable binding to a corresponding `(declare-const)` call, add an assertion that this SMT variable is equal to the bound expression and finally traverse each bound expression and the body expression, appending each translation wrapped in a scope to the resulting SMT program. We traverse the bound expressions separately from the actual variable binding as they may also contain calls that we wish to generate size-change assertions for.
- ▻ Finally, the (Case) rule handles `case` expressions, i.e. pattern matching. Similar to the translation of `let` expressions, we traverse the discriminating expression, i.e. the one that we are pattern-matching on, unlike the `let` translation we bind it to a fresh variable, however. For each branch we now generate a scoped assertion that the fresh variable we used for the discriminating expression is equal to the SMT translation of the pattern and then traverse the body expression of the branch.

[13]This might be considered a slight abuse of notation as the order of the assertions technically matters, but it should clarify that the assertions are combined in a way resembling a logical "and".

(Prim) $$e \Downarrow \varepsilon \qquad \text{where } e \text{ is a variable, literal or free declaration}$$

(Or) $$\frac{e_1 \Downarrow S_1 \quad e_2 \Downarrow S_2}{e_1 \textbf{ or } e_2 \Downarrow Scope(S_1) \land Scope(S_2)}$$

(Cons) $$\frac{x_1 \Downarrow S_1 \quad \dots \quad x_m \Downarrow S_m}{c(\overline{x_m}) \Downarrow \bigwedge_{1 \leqslant j \leqslant m} Scope(S_j)} \qquad \text{where } c \text{ is a constructor}$$

(Fun) $$\frac{x_1 \Downarrow S_1 \quad \dots \quad x_m \Downarrow S_m}{f(\overline{x_m}) \Downarrow \texttt{(echo "Call } f\texttt{")} \land \bigwedge_{1 \leqslant j \leqslant m} Scope(S_j) \land \bigwedge_{\substack{1 \leqslant i \leqslant n \\ 1 \leqslant j \leqslant m \\ w \in \{\downarrow \overline{\overline{\downarrow}}\}}} Scope(SizeChange(w, a_i, x_j))} \qquad \text{where } f \text{ is a function}$$

(Let) $$\frac{e_1 \Downarrow S_1 \quad \dots \quad e_m \Downarrow S_m \quad e \Downarrow S}{\textbf{let } \{\overline{x_m = e_m}\} \textbf{ in } e \Downarrow \bigwedge_{1 \leqslant j \leqslant m} (Bind(x_j, e_j) \land Scope(S_j)) \land Scope(S)}$$

(Case) $$\frac{e \Downarrow S \quad e_1 \Downarrow S_1 \quad \dots \quad e_m \Downarrow S_m}{\textbf{case } e \textbf{ of } \{\overline{p_m \to e_m}\} \Downarrow Scope(S) \land Bind(x, e) \land \bigwedge_{1 \leqslant j \leqslant m} Scope(Match(x, p_j) \land S_j)} \qquad \text{where } x \text{ is fresh}$$

where $\overline{a_n}$ refers to the enclosing function's arguments and the following operators are defined:

$$\begin{aligned}
Bind(x, e) &= \texttt{(declare-const } SMT(x) \ SMT(Type(e))\texttt{)(assert (= } SMT(x) \ SMT(e)\texttt{))} \\
Match(x, p) &= \texttt{(assert (= } SMT(x) \ SMT(p)\texttt{))} \\
Scope(S) &= \texttt{(push)} S \texttt{(pop)} \\
SizeChange(\downarrow, a, x) &= \texttt{(assert (} \geqslant \texttt{ (abs } x\texttt{) (abs } a\texttt{)))(check-sat)} \\
SizeChange(\overline{\overline{\downarrow}}, a, x) &= \texttt{(assert (} \neq \ x \ a\texttt{))(check-sat)} \\
Type(e) &\text{ denotes the type annotation of the FlatCurry expression } e \\
SMT(x) &\text{ denotes the SMT translation of the FlatCurry expression, type or pattern } x
\end{aligned}$$

Figure 5.3. SMT expression translation semantics for the computation of size-change graphs

Perhaps interestingly, we can handle both rigid and flexible matching in the same manner: Since `case` expressions are sequential, we could, in principle, assert the negation of all previous patterns in each branch. We do not need to do this, however, as FlatCurry already normalizes case expressions to only match a single level of constructors in each case expression and also ensures that pattern matching is complete, i.e. that every constructor has a branch.

As with the formal semantics described in chapter 3, the actual implementation deviates from the semantics slightly where it makes things more practical. For example, instead of just outputting the function name for each call, we output the `Show` notation of the `Call` to make it easily parsable and additionally echo the **Edge** for each combination of function and call argument, i.e. before generating the two size-change assertions, as that allows us to parse the output directly into a size-change graph, without having to reconstruct the precise orderings of the function and call arguments of each call.

Complete examples of how both the aforementioned factorial function and a Fibonacci function are translated to SMT, along with the outputs they produce when run through an SMT solver like Z3, can be found in appendix D.

Parsing the Output As mentioned earlier, the output of the SMT program is designed to be easily parsable into a size-change graph representation. Consider, for example, the Z3 output of the factorial function's SMT translation[14]:

```
Call {cSrcFunc = ("Factorial","fac"), cDstFunc = ("Factorial","fac")}
Edge {eSrcParam = 0, eDstParam = 0}
unsat
sat
Call {cSrcFunc = ("Factorial","fac"), cDstFunc = ("Prelude","_impl#*#Prelude.Num#Prelude.Int#")}
Edge {eSrcParam = 0, eDstParam = 0}
sat
unsat
```

We can parse the output without lookahead simply by interpreting the lines: Each call starts with a line containing the serialized **Call**, which is followed by zero or more edges. Each edge starts with a line containing the serialized **Edge** and is followed by exactly two lines, containing the negated result for the $\downarrow$ and $\overline{\downarrow}$ size relations, respectively. Parsing each call into a **Graph** is then a simple matter of constructing a **Map** from the edges and putting together the value. In our example, we would get the following graphs[15]:

```
[ Graph { gCall  = Call ("Factorial","fac") ("Factorial","fac")
        , gEdges = Map.fromList [(Edge 0 0, SmallerSize)]
        }
, Graph { gCall  = Call ("Factorial","fac") ("Prelude","_impl#*#Prelude.Num#Prelude.Int#")
        , gEdges = Map.fromList [(Edge 0 0, EqualSize)]
        }
]
```

Implementation of SMT-Based Graphs Like for the subterm-based strategy, we implement the SMT-based size-change graph generation monadically. Our implementation is stateful due to the need for fresh variables during the transformation, hence we define **SMTState**:

```
newtype SMTState = SMTState { smsNextVar :: Int }
```

As mentioned earlier, our implementation uses the `Curry2SMT` module known from the contract prover or non-failure verifier[16], which we import qualified as **SMT**. The monad that we use for the SMT translation, **SMTM**, is then defined as follows:

[14]As mentioned above, the full example can be found in section D.1

[15]We intentionally use the short notation for **Call** and **Edge** values to improve readability.

[16]We do, however, include a few tweaks to the vendored `Curry2SMT` module to support `(echo)`, `(push)`, `(pop)` and an updated variable translation scheme in order to support different prefixes. While this is an implementation detail, it lets us translate fresh variables using the `tmp` prefix, while normal FlatCurry variables keep using the `x` prefix, so we do not have to rename them on-the-fly during the translation.

```
type SMTM = StateT SMTState (ReaderT (VTFuncEnv TerminationInfo) (Writer [SMT.Command]))
```

This transformer includes the `SMTState` for the generation of fresh variables, a reader for providing access to the verification environment and a writer for emitting the SMT program. The equivalent to the subterm-based `graphsForFunc` is now `smtGraphsForFunc`, which has the following signature:

```
smtGraphsForFunc :: Options -> VTFuncEnv TerminationInfo -> TAFuncDecl -> VM [Graph]
```

Internally, this invokes `trFuncToSMT :: TAFuncDecl -> SMTM ()`, which takes care of the actual translation, pretty-prints the output as SMT-LIB code and feeds it to Z3. The Z3 output is then parsed to a `[Graph]` using the scheme described in the previous section. For the parsing itself we use a custom implementation of Parsec-style combinators that let us elegantly combine both line-based and character-based parsing using subparsers. A more extensive discussion of these parser combinators and how they apply here can be found in appendix E.

Merging the Subterm-Based and SMT-Based Graphs Since each of the strategies for constructing size-change graphs outputs a separate `[Graph]`, we need to bring these together. Unfortunately, we cannot just concatenate the lists, because both order and duplicates matter: Since a function may contain multiple calls to the same function with different arguments, a computed `[Graph]` may also contain multiple `Graph`s with the same `Call`. Now consider the case where our function has a single call and the subterm-based strategy computes an empty `Graph`, while the SMT-based strategy finds a decreasing edge. If we were to naively concatenate the graph lists, the result would effectively be interpreted as *two* calls where no size relationships could be established for the former and only the latter contains a decreasing argument. Since not every graph has a decreasing edge, the function would be labeled possibly looping even though our SMT strategy would have proven it to terminate.

For this reason we choose a different approach: The key insight here is that both `[Graph]`s are in the same order, thus the problem is similar to how two sorted arrays would be merged, e.g. in merge sort. Our algorithm effectively traverses both lists in linear time, while merging the two graphs when the calls match:

```
mergeGraphs :: [Graph] -> [Graph] -> [Graph]
mergeGraphs []       gs     = gs
mergeGraphs (sg:sgs) []     = sg : sgs
mergeGraphs (sg:sgs) (g:gs) = if gCall sg == gCall g then merge sg g : mergeGraphs sgs gs
                                                     else g : mergeGraphs (sg:sgs) gs
```

where the `merge` function combines two graphs by always picking the smaller edge:

```
merge :: Graph -> Graph -> Graph
merge (Graph c1 es1) (Graph c2 es2) =
  if c1 == c2
    then Graph c1 . deduplicateEdges . Map.toList $ Map.union es1 es2
    else error $ "Cannot merge different calls: " ++ ppCall c1 ++ ", " ++ ppCall c2
```

The heart of this function, `deduplicateEdges`, makes use of the fact that `Map.fromList` is right-biased, i.e. picks the last element from the list if multiple elements have the same key, to filter and construct a map that associates the smallest size relation for each edge in $O(n \log n)$ time:

```
deduplicateEdges :: [(Edge, SizeChange)] -> Map Edge SizeChange
deduplicateEdges = Map.fromList . sortBy ((>) `on` snd)
```

Note that the merge operator is distinct from the graph composition operator ";", whose implementation we will discuss later, as merge combines two graphs representing the *same* call whereas the composition operator takes a *sequence* of two calls, which are often different[17].

In the implementation of initTerminationInfo, the mergeGraphs function is invoked on both the subterm-based graphs and the smtgraphs, then wrapped in a TerminationInfo, together with the Unknown termination behavior, and returned. Thus, the initial verification value associated with each function contains both the size-change graphs and the Unknown termination behavior, the latter of which we will deal with in the next phase.

Iteration Phase: Composing the Graphs and Finding Threads of Infinite Descent

In this phase we perform the fixed-point iteration where the TerminationInfo of each function is updated until there are no changes in the associated values. This corresponds to the Iterate Verification Info stage described in section 4.4.2.

The algorithm implements the graph-based characterization of the size change principle introduced in section 5.2.3 and is based on the description by Lee et al. from [LJBA01]. At a high level, the idea is to build the transitive closure over the size-change graphs with respect to graph composition by repeatedly composing each pair of graphs until the set no longer changes. This maps nicely to our fixed-point iteration as the verification framework already provides us with the necessary infrastructure, so that we only have to implement the iteration step itself.

A prerequisite for this implementation is the graph composition operator, which we introduced as ";" to be consistent with the original paper. Since Haskell already uses semicolons for layout, we cannot name this operator (;), however, therefore we choose the name (>.>) instead, to reflect the left-to-right nature of the operator[18]. We implement the operator as follows:

```haskell
(>.>) :: Graph -> Graph -> Graph
(>.>) (Graph c1 es1) (Graph c2 es2) =
  if cDstFunc c1 == cSrcFunc c2
    then Graph
      { gCall  = Call (cSrcFunc c1) (cDstFunc c2)
      , gEdges =
          deduplicateEdges
            [ (Edge sp1 dp2, min sc1 sc2)
            | (Edge sp1 dp1, sc1) <- Map.toList es1
            , (Edge sp2 dp2, sc2) <- Map.toList es2
            , dp1 == sp2
            ]
      }
    else error $ "Cannot compose destination function " ++ ppName (cDstFunc c1)
                             ++ " with source function " ++ ppName (cSrcFunc c2)
```

As described in section 5.2.3, the idea of graph composition is that the composed graph represents the composition of the operands' calls, i.e. Graph c1 es1 >.> Graph c2 es2 represents the graph that results from first performing c1 and then c2. The implementation is relatively straightforward and

[17]Composing the same call twice only works if the call is recursive, which would then be interpreted as two nested invocations of the function.

[18]Note that left-to-right order is the reverse of mathematical function composition or application, which is generally written right-to-left. The operator name (>.>) is intentionally chosen to resemble Haskell's (>>>) from Control.Arrow, which provides a generic left-to-right composition operator, which can also be used for reverse function composition.

begins with a check that the calls can indeed be composed, by verifying that the destination function of the left call matches the source function of the right call. If this is the case, we create a composite call from the left source function to the right destination function and compute the edges by taking the Cartesian product of the left and the right edges, filtered by calls that match, and deduplicating the edges using the same mechanism that we implemented for the `merge` function. In other words, we pick the smaller size relation of every pair of matching calls. As mentioned earlier, the composition operator `(>.>)` is associative, so we can omit parentheses.

To provide some examples, we will introduce a small convenience function that simplifies the notation of constructing a simple `Graph`:

```
mkGraph :: String -> String -> [(Edge, SizeChange)] -> Graph
mkGraph f g es = Graph (mkCall f g) (Map.fromList es)
  where mkCall f g = Call (mkName f) (mkName g)
        mkName n   = ("", n)
```

Composing two trivial graphs, e.g. `mkGraph "f" "g" [] >.> mkGraph "g" "h" []`, yields an empty graph, in this case `mkGraph "f" "h" []`, since there are no edges to consider. In fact, if either the left-hand or right-hand side has no edges, the resulting graph will not have any edges either. Intuitively this makes sense as there is no thread of size-change relations passing through both calls. If both graphs include edges that together form a thread, the resulting graph will contain an edge for each such thread, labeled with the smaller of the two size-change relations. For example,

```
mkGraph "f" "g" [(Edge 0 1, SmallerSize)] >.> mkGraph "g" "h" [(Edge 1 2, EqualSize)]
```

would simplify to

```
mkGraph "f" "h" [(Edge 0 2, SmallerSize)]
```

For a slightly more practical example, recall the `take` function introduced in section 5.2.2 and illustrated in figure 5.1. The composition of these graphs could be expressed as follows

```
mkGraph "take"  "takep" [(Edge 0 0, EqualSize), (Edge 1 1, EqualSize)] >.>
mkGraph "takep" "take"  [(Edge 1 1, SmallerSize)]
```

which would evaluate to

```
mkGraph "take"  "take"  [(Edge 1 1, SmallerSize)]
```

The iteration step proceeds in two parts, computing the graphs and the termination behavior of the new `TerminationInfo`, respectively:

1. To compute the graphs, we take every combination of an existing graph starting at the current function and a call from the function's body expression, where the graph's destination function matches the called function, and compose it with every existing graph from the called function. Intuitively this amounts to going one call level deeper with every iteration, until the set of graphs no longer changes and we have the full transitive closure over the size-change graphs.

2. To compute the termination behavior, we look for all cyclic graphs, i.e. graphs of the form `Graph (Call f f) ...`, that are simultaneously idempotent, i.e. graphs `g` for which `g >.> g == g`. Since we draw the graphs from the transitive closure, we are intuitively considering all cyclic *call sequences*. If any of these cyclic idempotent graphs end at a possibly looping function, i.e. intuitively if the current function can reach a possibly looping function via some call sequence, we output

PossiblyLooping. If not, we proceed to check for threads of infinite descent. We can identify a thread of infinite descent by looking for an edge of the form `Edge i i` that is strictly decreasing, i.e. is associated a SmallerSize, among the cyclic idempotent graphs. If we can find a thread of infinite descent for every cyclic idempotent graph, i.e. cyclic call sequence, we output Terminating, otherwise we output PossiblyLooping, since we cannot prove that every recursion decreases some argument.

Since we assume all primitives terminate, we also mark the function as trivially Terminating if it is external. This covers a variety of functions from the Prelude, including e.g. implementations of numeric operators, I/O and the failed function.

The implementation of this stage is located in analyzeFuncTermination, whose signature is derived from vAnalyze and can be expressed as follows:

```
analyzeFuncTermination :: VTFuncEnv TerminationInfo -> VM (VTFuncUpdate TerminationInfo)
```

In this function, we fetch the current function declaration, along with the previously computed verification values, i.e. TerminationInfo values containing size-change graphs for all known functions, and then compute the updated TerminationInfo using the scheme described above.

5.2.6 Examples

In this section we will illustrate our size-change principle-based termination checker on a few more example programs, showcasing its features, how it compares to CASS and what its limitations are.

Trivial Recursion

```
loop :: a
loop = loop
```

This function is recognized by both CASS and our termination checker as possibly looping. Since neither the subterm, nor the SMT-based strategy can find a decreasing edge, we get the following justification:

```
loop -> PossiblyLooping ["Cyclic call sequences without threads of infinite descent in loop"]
```

Nondeterminism

```
coin :: Bool
coin = True ? False
```

Nondeterminism does not affect a function's termination behavior: We traverse the arguments of a nondeterministic function like the choice operator like any other operator. Both CASS and our termination checker are therefore able to prove that coin terminates:

```
coin -> Terminating ["No cyclic call sequences"]
```

Direct Recursion

```
last :: [a] -> a
last [x]      = x
last (_:_:xs) = last xs
```

```
aNumber :: Int
aNumber = last [42, 24]
```

The `last` function terminates, since for any finite[19] list the `last` function will eventually call `last []`, and the `aNumber` value terminates since it only calls terminating functions and is non-recursive. Indeed, both CASS and our termination checker are able to prove these two functions as terminating, since `last` only contains a directly recursive call on a direct subterm of its argument. Our termination checker outputs the following justifications:

```
aNumber -> Terminating ["No cyclic call seqs."]
last    -> Terminating ["Each cyclic call sequence has a thread of infinite descent: {last: 0}"]
```

Indirect (Mutual) Recursion

```
even :: [a] -> Bool
even []     = True
even (_:xs) = odd xs

odd :: [a] -> Bool
odd [_]        = True
odd (_:_:_:xs) = even xs
```

While CASS is unable to prove that either one of these functions terminates and outputs `possibly non-terminating` for both functions, our termination checker finds a thread of infinite descent for both functions, thus correctly labels them as terminating:

```
even -> Terminating ["Each cyclic call sequence has a thread of infinite descent: {even: 0}"]
odd  -> Terminating ["Each cyclic call sequence has a thread of infinite descent: {odd: 0}"]
```

Take and Split

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

splitAt :: Int -> [a] -> ([a], [a])
splitAt n l = if n <= 0 then ([], l) else splitAtp n l
  where splitAtp _ []     = ([], [])
        splitAtp m (x:xs) = let (ys, zs) = splitAt (m - 1) xs in (x : ys, zs)
```

Like in the previous case, CASS is unable to prove that either one of these functions terminate while our termination checker can recognize the strict descent in the list argument to prove termination in both cases. Here the checker produces the following output, which includes selector functions generated by FlatCurry[20]:

```
splitAt -> Terminating ["Each cyclic call sequence has a thread of infinite descent: {splitAt: 1}"]
take    -> Terminating ["Each cyclic call sequence has a thread of infinite descent: {take: 1}"]
```

[19] Recall that we do not consider infinite lists in our definition of termination given that infinite lists do not have an NF.

[20] The actual output includes additional functions generated by FlatCurry, including `takep`, `splitAtp` and some selectors. We omit these for readability, they are similarly marked as `Terminating`, however.

Higher-Order Functions

```
map :: (a -> b) -> [a] -> [b]
map _ []     = []
map f (x:xs) = f x : map f xs

foldr :: (a -> b -> b) -> b -> [a] -> b
foldr _ y []     = y
foldr f y (x:xs) = f x (foldr f y xs)
```

Higher-order functions like `map` or `foldr` are treated like any other directly recursive function, thus both CASS and our termination checker are able to prove these as terminating:

```
foldr  -> Terminating ["Each cyclic call sequence has a thread of infinite descent: {foldr: 2}"]
length -> Terminating ["Each cyclic call sequence has a thread of infinite descent: {length: 0}"]
```

Non-Shrinking Recursion

```
repeat :: a -> [a]
repeat x = x : repeat x
```

If there are recursive calls where no argument is strictly decreasing, our termination criterion does not apply. Here, both CASS and our termination checker correctly prove that `repeat` does not terminate:

```
repeat  -> PossiblyLooping ["Cyclic call sequences without threads of infinite descent in repeat"]
```

Infinite Data Structures

```
xs :: [Int]
xs = map (*2) (repeat 1)

headOfXs :: Int
headOfXs = head xs

infinity :: Int
infinity = length xs
```

Neither CASS nor our termination checker are able to prove any of these values as terminating, in our case we get the following justification:

```
xs       -> PossiblyLooping ["Calls to possibly looping functions Prelude.repeat"]
headOfXs -> PossiblyLooping ["Calls to possibly looping functions xs, Prelude.repeat"]
infinity -> PossiblyLooping ["Calls to possibly looping functions xs, Prelude.repeat"]
```

This highlights an important limitation of our termination checker, namely that the notion of NF-termination which we operate on effectively assumes strictness and therefore cannot prove termination for any functions involving infinite data structures, even if they terminate as `headOfXs` would due to Curry's lazy evaluation.

Growing Argument Values

```
data Peano = Zero | Succ Peano

leq :: Peano -> Peano -> Bool
leq p1 p2 = case (p1, p2) of
  (Zero    , _       ) -> True
  (Succ _  , Zero    ) -> False
  (Succ p1', Succ p2') -> leq p1' p2'
```

Unlike CASS, our termination checker can successfully infer the strictly decreasing size relation between `p1` and `p1'`, as well as `p2` and `p2'`, to prove termination:

```
leq -> Terminating ["Each cyclic call sequence has a thread of infinite descent: {leq: 0}"]
```

Factorial

```
fac'pre :: Int -> Bool
fac'pre n = n >= 0

fac :: Int -> Int
fac n = if n == 0 then 1
                  else n * fac (n - 1)
```

While CASS is unable to prove either of these functions as terminating, interestingly not even `fac'pre`, our termination checker handles both, thanks to our SMT-based size change graph construction strategy which lets us prove $|n-1| < |n|$:

```
fac     -> Terminating ["Each cyclic call sequence has a thread of infinite descent: {fac: 0}"]
fac'pre -> Terminating ["No cyclic call sequences"]
```

As mentioned earlier, the SMT translation for this function can be found in appendix D.

Fibonacci

```
fib'pre :: Int -> Bool
fib'pre n = n >= 0

fib :: Int -> Int
fib n = case n of
  0 -> 0
  1 -> 1
  _ -> fib (n - 2) + fib (n - 1)
```

Again, our termination checker is able to prove termination, unlike CASS:

```
fib     -> Terminating ["Each cyclic call sequence has a thread of infinite descent: {fib: 0}"]
fib'pre -> Terminating ["No cyclic call sequences"]
```

Likewise, the SMT translation can be found in appendix D.

5.2.7 Discussion

As demonstrated by the examples above, our termination checker can handle a variety of different cases, including indirect recursion or integer arithmetic. This stands in contrast to previous implementations like the CASS termination analysis, which can only handle simple cases of direct recursion. Still, a number of limitations remain: The SMT translation scheme that we use to compute size-change graphs based on the absolute value ordering cannot handle functions involving algebraic data types yet. This means that our SMT translation will currently only work for functions that are purely defined in terms of primitive types such as numbers of booleans, yet fail if algebraic types such as lists are involved. Although this is often not an issue in practice if the subterm-based strategy can provide a thread of infinite descent, it limits our ability to prove termination even in cases where an integer argument can be shown to shrink in terms of the absolute value ordering, simply because another, algebraic argument could not be translated. An example of this would be the following program, which constructs a binary tree of a certain height:

```haskell
data Tree a = Branch [Tree a] | Leaf a

binaryTree :: Int -> a -> Tree a
binaryTree n x =
  if n <= 0
    then Leaf x
    else Branch [t, t]
  where t = binaryTree (n - 1) x
```

Even though the function terminates, both in practice and under our definition, our checker outputs

```
PossiblyLooping ["Cyclic call sequences without threads of infinite descent in binaryTree"]
```

The problem is that the only strictly decreasing edge in the recursive call is the integer argument. However, since our SMT translation is unable to handle custom data types, such as `Tree a` in this case, it fails and does not yield any graphs, hence our checker cannot find a thread of infinite descent either. We will discuss potential strategies for mitigating this in section 6.2.4, but consider these ideas to be out-of-scope for this thesis.

From an implementation perspective, our size-change principle-based termination checker leverages the verification framework's fixed-point iteration model, whose division into initialization and iteration phases maps well to the split between computing and composing size-change graphs. This abstraction boundary provides a clear separation of concerns between the framework's traversal and fixed-point iteration over the function dependency graph and the termination checker's size-change graph logic, allowing both sides to be optimized independently. This has direct practical relevance as our framework is still comparatively slow on larger module graphs and would require framework-level optimizations to make these cases performant. Although caching helps with this, given that our fixed-point iteration is a relatively naive implementation that traverses all uncached modules in each iteration, running a framework-based verification like the termination checker on a larger project can still be too slow to be practical. Fortunately, there are some relatively simple strategies that could be implemented in the framework for this, as we will see in section 6.2.1. We will leave these for the future too, however, to maintain our focus on smaller examples.

Chapter 6

Conclusion

To sum up the thesis, we will provide a recap of our implementation, summarize our results and sketch out a number of ideas for future work.

6.1 Summary and Results

A unified framework for implementing program verifications in Curry has proven to be a powerful abstraction that takes much of the heavy lifting involved in traversing, transforming and caching programs in FlatCurry, the intermediate language emitted by the Curry compiler frontend. This includes function-level dependency tracking, generic simplifying transformations and centralized infrastructure for fixed-point computation of verification-specific result values that are associated with each function. The port of the existing contract prover to a client of the framework shows that existing verifications can be adapted to the framework's infrastructure and a new, size-change principle termination checker showcases the framework's ability to provide a foundation for new verification tools. The termination checker additionally meaningfully generalizes the ideas behind the termination analysis of CASS, adding support for functions involving integer arithmetic by leveraging an SMT solver that verifies numeric size-change relations between the arguments of every function and its calls and mutual recursion by inferring size-change relations across functions.

Still, there are a number of ways in which both the framework and the termination checker could be improved, generalized or optimized. Many of these optimizations can be applied directly at the framework-level, thus allowing these improvements to be shared across all tools based on the framework. This includes e.g. optimizing the dependency tracking by computing *Strongly Connected Components (SCCs)* or the caching mechanism via a more compact data representation. In the termination checker, the SMT translation scheme could be improved to add support for structured data types, which would allow for even more programs to be verified as terminating. We will discuss these ideas briefly in the following section.

6.2 Future Work

As an inspiration for future work, this section contains a list of ideas for how the verification framework could be extended, improved or built upon. This is by no means an exhaustive list, but may serve as a starting point for further research or development of the framework and verifications in Curry.

6.2.1 Optimizing the Iteration with Working Lists and Toposorted SCCs

Currently, the verification framework implements a relatively simple strategy for the fixed-point iteration, namely to just iterate the `vAnalyze` function over all tracked functions until nothing changes anymore. This is comparatively inefficient and there are several optimizations to deal with this that

CASS, for example, has implemented, but are still missing in the framework. One such optimization is the use of *working lists*: Instead of iterating on all tracked functions, we could compute the transitive closure of callees over all updated functions and then continue the iteration only on this set of "dirty" functions that need to be updated again. All other functions are "clean", as none of their dependencies changed, therefore it would be safe to skip them in all following iterations, as we have already found their fixed point. This can be optimized even further via the use of SCCs, as we can compute a topological ordering over the SCCs on the global function dependency graph initially and then perform individual fixed-point iterations on each SCC. This way, we would only iterate on groups of mutually recursive functions, which would likely provide considerable performance improvements as much fewer functions would need to be iterated over, especially if the SCCs are not very large.

6.2.2 Improving Caching with a More Efficient Data Representation

Currently, the framework uses the standard `Read` and `Show` type classes for serializing and deserializing data, including various containers and the client-provided verification info, as described in section 4.5. Recent work by Züngel has shown that a more compact representation of Curry values, named `rw-data`, can improve performance by a factor of 100 over the standard `Read` instances when deserializing Peano numbers and even provide a considerable improvement over `ReadShowTerm`, a module that reimplements the standard `read` function as an external operation with a native implementation for each Curry compiler [Zün24]. Given that I/O has proven to be a bottleneck, especially in interpreted Curry environments such as PAKCS, integrating `rw-data` into our caching infrastructure, similar to CASS, could provide meaningful performance improvements to verifications implemented using our framework. It should be noted that adding the necessary `ReadWrite` constraint to the verification value type parameter `a` in the framework's entry points would constitute a source break as all clients of the framework would be required to add `ReadWrite` instances to their verification value types, though these instances can be generated automatically using the `rw-data-generator` tool.

6.2.3 Exposing More Metadata via JSON and Implementing an IDE Integration

As mentioned in section 4.6, scriptability unlocks a range of new ways in which verification tools can be automated or embedded. One such use case would be integrating into existing development infrastructure, specifically by adding an IDE integration, e.g. for Visual Studio Code, Emacs or Neovim. The *Language Server Protocol (LSP)* [Mic17] provides an abstraction between the editor and language features such as hover, jump-to-definition or code completion and the Curry language server [Wie20] provides an LSP server implementation for Curry. A recently added extensibility API in the Curry language server makes it possible to add custom hovers through external tools that are invoked via user-defined shell commands, taking metadata about the current editing context[1]. In conjunction with the verification framework's JSON output format as discussed in section 4.6.1, this could be used to implement adapters for various verification tools by exposing e.g. the call type or termination info of the hovered operation. Alternatively, warnings or inline editor decorations could be generated that update live as the user types[2]. Depending on how this information is presented to the user, the framework may have to provide additional metadata in its JSON output, such as the source locations of the verified functions, ideally in a configurable way that is minimally invasive in terms of breaking changes to the framework's API.

[1]See `https://github.com/fwcd/curry-language-server/pull/77` for details.

[2]Performance might be an issue, especially in larger projects, but with efficient caching and other tricks such as debouncing it might be feasible to implement this in a snappy way.

6.2.4 Generalizing the SMT-Based Termination Checker

The SMT-based size-change graph construction strategy of the termination checker described in section 5.2 is well-suited for Curry functions involving integer arithmetic and preconditions. There are, however, still a number of additions that could make it more useful in a wider variety of cases. One possible extension would be implementing support for postconditions to provide additional information to the SMT solver around called functions. Additionally, the number of supported data types could be extended. Currently, the SMT translation scheme only handles primitive types such as integers, booleans, strings or similar types that have directly equivalent SMT sorts. The `Curry2SMT` module that we vendor from the contract prover and non-failure verifier already provides a function for translating Curry data types to SMT, which we could use to generate corresponding SMT declarations. One way to integrate this would be analyzing the function for all types it depends on, including e.g. both custom or `Prelude` types such as tuples and lists, and then prepending the translated SMT declarations to the program. This would let the SMT-based size change graph strategy handle functions mixing integer arithmetic with algebraic data structures, which it does not support yet.

6.2.5 Porting the Non-Failure Verification Tool

Besides the contract prover, the non-failure verification tool introduced in section 2.2.3 could be ported to the framework. This will likely prove to be a more challenging task than the contract prover as the tool manages a lot of internal state and implements its own mechanisms for program-level caching of a number of intermediate results, including analysis information about constructors and previously computed I/O types, call types and non-failure conditions. A port of the non-failure verification tool could wrap these in a structure that constitutes the computed verification info, thereby taking advantage of the framework's caching infrastructure:

```
data VerifyInfo a = VerifyInfo
  { viNonFailCond :: Maybe NonFailCond
  , viCallType    :: Maybe (ACallType a)
  , viIOType      :: Maybe (InOutType a)
  }
```

However, parts of the program that make assumptions about global state, e.g. by taking `IORef`s of program info, would have to be rewritten to leverage the framework's structured state handling instead. Additionally, all program-level operations would have to be either moved into the preprocessing stage or split up into function-level operations during the init or analyze stage.

6.2.6 Implementing a Determinism Checker

Considering the framework is just a set of building blocks, the space of new verifications that could be implemented using the framework is still largely unexplored. Other analyses implemented in CASS could be ported similarly to our termination checker, extended with new techniques such as SMT solving and take advantage of the framework's conveniences, such as efficient fine-grained, function-level caching. The determinism analysis is one such example, which has been well-studied in the past [HS00; BH05a; FKS11; AH17], partially also due to its direct application in KiCS2 where deterministic operations benefit from simpler code generation.

Bibliography

[AEH00] Sergio Antoy, Rachid Echahed, and Michael Hanus. “A needed narrowing strategy”. In: *Journal of the ACM* 47.4 (July 2000), pp. 776–822. DOI: 10.1145/347476.347484.

[AH12] Sergio Antoy and Michael Hanus. “Contracts and Specifications for Functional Logic Programming”. In: *Practical Aspects of Declarative Languages*. Ed. by Claudio Russo and Neng-Fa Zhou. Vol. 7149. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 33–47. DOI: 10.1007/978-3-642-27694-1_4.

[AH17] Sergio Antoy and Michael Hanus. “Eliminating Irrelevant Non-determinism in Functional Logic Programs”. In: *Practical Aspects of Declarative Languages*. Ed. by Yuliya Lierler and Walid Taha. Vol. 10137. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2017, pp. 1–18. DOI: 10.1007/978-3-319-51676-9_1.

[AHH+05] Elvira Albert, Michael Hanus, Frank Huch, Javier Oliver, and Germán Vidal. “Operational semantics for declarative multi-paradigm languages”. In: *Journal of Symbolic Computation* 40.1 (July 2005), pp. 795–829. DOI: 10.1016/j.jsc.2004.01.001.

[ALS+07] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman, eds. *Compilers: Principles, Techniques, and Tools*. 2. ed., Pearson internat. ed. Boston Munich: Pearson Addison-Wesley, 2007. 1009 pp. ISBN: 978-0-321-48681-3 978-0-321-49169-5.

[BH05a] Bernd Braßel and Michael Hanus. “Nondeterminism Analysis of Functional Logic Programs”. In: *Logic Programming*. Ed. by Maurizio Gabbrielli and Gopal Gupta. Vol. 3668. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 265–279. DOI: 10.1007/11562931_21.

[BH05b] Bernd Braßel and Frank Huch. “Translating Curry To Haskell System Demo”. In: *Proceedings of the 2005 ACM SIGPLAN workshop on Curry and functional logic programming*. WCFLP05: Workshop on Curry and Functional Logic Programming. Tallinn Estonia: ACM, Sept. 29, 2005, pp. 60–65. DOI: 10.1145/1085099.1085112.

[BHP+11] Bernd Braßel, Michael Hanus, Björn Peemöller, and Fabian Reck. “KiCS2: A New Compiler from Curry to Haskell”. In: *Functional and Constraint Logic Programming*. Ed. by Herbert Kuchen. Vol. 6816. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 1–18. DOI: 10.1007/978-3-642-22531-4_1.

[BST10] Clark Barrett, Aaron Stump, and Cesare Tinelli. “The SMT-LIB Standard – Version 2.0”. In: *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories* (July 2010). URL: http://theory.stanford.edu/~barrett/pubs/BST10.pdf.

[Coo71] Stephen A. Cook. “The complexity of theorem-proving procedures”. In: *Proceedings of the third annual ACM symposium on Theory of computing - STOC '71*. the third annual ACM symposium. Shaker Heights, Ohio, United States: ACM Press, 1971, pp. 151–158. DOI: 10.1145/800157.805047.

[Dav58] Martin Davis. *Computability & Unsolvability*. Dover Publications, 1958.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Vol. 4963. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24.

[FKS11] Sebastian Fischer, Oleg Kiselyov, and Chung-Chieh Shan. "Purely functional lazy nondeterministic programming". In: *Journal of Functional Programming* 21.4 (Sept. 2011), pp. 413–465. DOI: 10.1017/S0956796811000189.

[Fow05] Martin Fowler. *Inversion Of Control*. martinfowler.com. June 26, 2005. URL: https://martinfowler.com/bliki/InversionOfControl.html (visited on 06/14/2025).

[GGB+] Yitzchak Gale et al. *ListT done right - HaskellWiki*. URL: https://wiki.haskell.org/ListT_done_right (visited on 07/27/2025).

[GSJ+24] Lennard Gäher, Michael Sammler, Ralf Jung, Robbert Krebbers, and Derek Dreyer. "RefinedRust: A Type System for High-Assurance Verification of Rust Programs". In: *Proceedings of the ACM on Programming Languages* 8 (PLDI June 20, 2024), pp. 1115–1139. DOI: 10.1145/3656422.

[HAB+10] Michael Hanus et al. *PAKCS: The Portland Aachen Kiel Curry System*. 2010. URL: https://www.curry-lang.org/pakcs.

[Han13] Michael Hanus. "Functional Logic Programming: From Theory to Curry". In: *Programming Logics*. Ed. by Andrei Voronkov and Christoph Weidenbach. Vol. 7797. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 123–168. DOI: 10.1007/978-3-642-37651-1_6.

[Han16] Michael Hanus. *Curry: An Integrated Functional Logic Language*. Jan. 13, 2016. URL: https://www.curry-lang.org.

[Han17a] Michael Hanus. "Combining Static and Dynamic Contract Checking for Curry". In: *Logic-Based Program Synthesis and Transformation*. Ed. by Fabio Fioravanti and John P. Gallagher. Vol. 10855. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2017, pp. 323–340. DOI: 10.1007/978-3-319-94460-9_19.

[Han17b] Michael Hanus. *The flatcurry-annotated package*. May 17, 2017. URL: https://github.com/curry-packages/flatcurry-annotated (visited on 08/11/2025).

[Han17c] Michael Hanus. *The flatcurry package*. Sept. 26, 2017. URL: https://github.com/curry-packages/flatcurry (visited on 08/11/2025).

[Han18] Michael Hanus. "Verifying Fail-Free Declarative Programs". In: *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming*. PPDP '18: The 20th International Symposium on Principles and Practice of Declarative Programming. Frankfurt am Main Germany: ACM, Sept. 3, 2018, pp. 1–13. DOI: 10.1145/3236950.3236957.

[Han24] Michael Hanus. "Inferring Non-failure Conditions for Declarative Programs". In: *Functional and Logic Programming*. Ed. by Jeremy Gibbons and Dale Miller. Vol. 14659. Series Title: Lecture Notes in Computer Science. Singapore: Springer Nature Singapore, 2024, pp. 167–187. DOI: 10.1007/978-981-97-2300-3_10.

[Han25] Michael Hanus. "Hybrid Verification of Declarative Programs with Arithmetic Non-fail Conditions". In: *Programming Languages and Systems*. Ed. by Oleg Kiselyov. Vol. 15194. Series Title: Lecture Notes in Computer Science. Singapore: Springer Nature Singapore, 2025, pp. 109–129. DOI: 10.1007/978-981-97-8943-6_6.

[HHPJ+07] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. "A history of Haskell: being lazy with class". In: *Proceedings of the third ACM SIGPLAN conference on History of programming languages*. HOPL-III '07: ACM SIGPLAN History of Programming Languages Conference III. San Diego California: ACM, June 9, 2007. DOI: 10.1145/1238844.1238856.

[HM96] Graham Hutton and Erik Meijer. *Monadic Parser Combinators*. University of Nottingham, 1996.

[HS00] Michael Hanus and Frank Steiner. "Type-based nondeterminism checking in functional logic programs". In: *Proceedings of the 2nd ACM SIGPLAN international conference on Principles and practice of declarative programming*. PPDP00: International Conference on Principles and Practice of Declarative Programming. Montreal Quebec Canada: ACM, Sept. 2000, pp. 202–213. DOI: 10.1145/351268.351292.

[HS14] Michael Hanus and Fabian Skrlac. "A modular and generic analysis server system for functional logic programs". In: *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation*. POPL '14: The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. San Diego California USA: ACM, Jan. 11, 2014, pp. 181–188. DOI: 10.1145/2543728.2543744.

[Jel18] Wolfgang Jeltsch. *The Difference Between Head Normal Form (HNF) and Weak Head Normal Form (WHNF)*. r/haskell. Nov. 21, 2018. URL: https://www.reddit.com/r/haskell/comments/9z6v51/whats_the_difference_between_head_normal_formhnf/ea6vuz3/ (visited on 09/13/2025).

[Kar72] Richard Karp. "Reducibility Among Combinatorial Problems". In: Raymond E. Miller. *Complexity of Computer Computations*. Ed. by James W. Thatcher. New York: Plenum, 1972, pp. 85–103. ISBN: 0-306-30707-3. URL: https://www.cs.purdue.edu/homes/hosking/197/canon/karp.pdf (visited on 09/23/2025).

[Kin19] Alexis King. *Parse, don't validate*. Nov. 5, 2019. URL: https://lexi-lambda.github.io/blog/2019/11/05/parse-don-t-validate/ (visited on 09/06/2025).

[KNK18] Steve Klabnik, Carol Nichols, and Chris Krycho. *The Rust Programming Language*. No Starch Press, 2018. 488 pp. ISBN: 978-1-59327-828-1. URL: https://doc.rust-lang.org/book/ (visited on 10/09/2025).

[Krü21] Leif-Erik Krüger. "Erweiterung von Curry um Multiparametertypklassen". Master's Thesis. Kiel University, Oct. 2021.

[Lau93] John Launchbury. "A natural semantics for lazy evaluation". In: *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '93*. the 20th ACM SIGPLAN-SIGACT symposium. Charleston, South Carolina, United States: ACM Press, 1993, pp. 144–154. DOI: 10.1145/158511.158618.

[LHJ95] Sheng Liang, Paul Hudak, and Mark Jones. "Monad transformers and modular interpreters". In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '95*. the 22nd ACM SIGPLAN-SIGACT symposium. San Francisco, California, United States: ACM Press, 1995, pp. 333–343. DOI: 10.1145/199448.199528.

[LJBA01] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. "The size-change principle for program termination". In: *ACM SIGPLAN Notices* 36.3 (Mar. 2001), pp. 81–92. DOI: 10.1145/373243.360210.

[LM01] Daan Leijen and Erik Meijer. *Parsec: Direct Style Monadic Parser Combinators For The Real World*. UU-CS-2001-27. Utrecht University, Oct. 4, 2001. URL: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/parsec-paper-letter.pdf.

[Luc21] Salvador Lucas. "The origins of the halting problem". In: *Journal of Logical and Algebraic Methods in Programming* 121 (June 2021), p. 100687. DOI: 10.1016/j.jlamp.2021.100687.

[McC12] C. A. McCann. *Answer to "Why is there no IO transformer in Haskell?"* Stack Overflow. Oct. 24, 2012. URL: https://stackoverflow.com/a/13057015/19890279 (visited on 07/27/2025).

[Mic17] Microsoft. *Language Server Protocol*. 2017. URL: https://microsoft.github.io/language-server-protocol/ (visited on 09/19/2025).

[Mit04] Neil Mitchell. "Termination checking for a lazy functional language". Presentation from my first year literature review seminar. Dec. 21, 2004. URL: https://ndmitchell.com/downloads/slides-termination_checking_for_a_lazy_functional_language-21_dec_2004.pdf (visited on 09/12/2025).

[Obe16a] Jonas Oberschweiber. "A Package Manager for Curry". Master's Thesis. Sept. 2016. URL: https://www.informatik.uni-kiel.de/~mh/lehre/abschlussarbeiten/msc/Oberschweiber.pdf (visited on 12/15/2020).

[Obe16b] Jonas Oberschweiber. *The json package*. Apr. 2016. URL: https://github.com/curry-packages/json (visited on 09/19/2025).

[Pau22] Srijan Paul. *Monadic parser combinators in Haskell*. DeepSource. June 9, 2022. URL: https://deepsource.com/blog/monadic-parser-combinators (visited on 10/03/2025).

[PJ87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. 10. [pr.] Prentice-Hall International series in computer science. New York: Prentice-Hall, May 1987. 445 pp. ISBN: 978-0-13-453333-9 978-0-13-453325-4.

[Ses02] Peter Sestoft. "Demonstrating Lambda Calculus Reduction". In: *The Essence of Computation*. Ed. by Torben Æ. Mogensen, David A. Schmidt, and I. Hal Sudborough. Vol. 2566. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 420–435. DOI: 10.1007/3-540-36377-7_19.

[Sla74] James R. Slagle. "Automated Theorem-Proving for Theories with Simplifiers Commutativity, and Associativity". In: *Journal of the ACM* 21.4 (Oct. 1974), pp. 622–642. DOI: 10.1145/321850.321859.

[Tor22] Linus Torvalds. *Initial Rust support in the Linux kernel*. Oct. 3, 2022. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=8aebac82933ff1a7c8eede18cab11e1115e2062b (visited on 10/09/2025).

[Tur36] Alan Turing. "On Computable Numbers, with an Application to the Entscheidungsproblem". In: *Proceedings of the London Mathematical Society* 2.42 (1936), pp. 230–265. URL: https://people.math.ethz.ch/~halorenz/4students/Literatur/TuringFullText.pdf (visited on 09/09/2025).

[VSJ14] Niki Vazou, Eric L. Seidel, and Ranjit Jhala. "LiquidHaskell: experience with refinement types in the real world". In: *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell*. ICFP'14: ACM SIGPLAN International Conference on Functional Programming. Gothenburg Sweden: ACM, Sept. 3, 2014, pp. 39–51. DOI: 10.1145/2633357.2633366.

[VSJ+14] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. "Refinement types for Haskell". In: *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming*. ICFP'14: ACM SIGPLAN International Conference on Functional Programming. Gothenburg Sweden: ACM, Aug. 19, 2014, pp. 269–282. DOI: 10.1145/2628136.2628161.

[Wad90] Philip Wadler. “Comprehending monads”. In: *Proceedings of the 1990 ACM conference on LISP and functional programming*. LFP90: ACM Conference on Lisp and Functional Programming. Nice France: ACM, May 1990, pp. 61–78. DOI: 10.1145/91556.91592.

[Wie20] Fredrik Wieczerkowski. *Curry Language Server*. 2020. URL: https://github.com/fwcd/curry-language-server (visited on 09/19/2025).

[WPP77] David H D Warren, Luis M. Pereira, and Fernando Pereira. “Prolog - the language and its implementation compared with Lisp”. In: *ACM SIGART Bulletin* 64 (Aug. 1977), pp. 109–115. DOI: 10.1145/872736.806939.

[Zün24] Lasse Züngel. “Kompakte Repräsentation von Datentermen”. Bachelor’s Thesis. Kiel University, Apr. 2024. URL: https://www.michaelhanus.de/lehre/abschlussarbeiten/bsc/Zuengel_Lasse.pdf (visited on 06/19/2025).

Acronyms

API Application Programming Interface. i, 17, 18, 22, 23, 27, 28, 30, 34, 41, 42, 72, 87, 89

BNF Backus-Naur-Form. i, 9

CASS Curry Analysis Server System. i, v, 2, 17–19, 22, 39, 40, 45, 48, 50, 56, 57, 66–73

CI Continuous Integration. i, 41

CPM Curry Package Manager. i, 11, 27

CTC Call-Time Choice. i, 4–6

DFS Depth-First Search. i, 33, 41

HNF Head Normal Form. i, 5

I/O Input/Output. i, 1, 2, 6–8, 15–18, 24, 35, 38, 39, 57, 66, 72, 73, 90

IDE Integrated Development Environment. i, 41, 72

IoC Inversion of Control. i, 22

JSON JavaScript Object Notation. i, 41–43, 48, 72

KiCS2 Kiel Curry System Version 2. i, 4, 8, 9, 12, 73

LSP Language Server Protocol. i, 72

MPTC Multi-Parameter Type Class. i, 8, 22

NF Normal Form. i, 5, 36, 49, 67, 68

PAKCS Portland Aachen Kiel Curry System. i, 4, 8, 9, 12, 39, 40, 72

REPL Read-Eval-Print-Loop. i

RTC Run-Time Choice. i, 4

SAT Boolean Satisfiability Problem. i, 12, 13

SCC Strongly Connected Component. i, 71, 72

SMT Satisfiability Modulo Theories. i, v, ix, 2, 12–18, 21, 24, 34–36, 46–48, 55, 58–63, 66, 69–71, 73, 91, 92, 95, 97

TRS Term Rewriting System. i, 3

WHNF Weak Head Normal Form. i, 5, 6, 11

Appendix A

FlatCurry Syntax Types

Below is the full definition of both untyped and (type-)annotated FlatCurry syntax in terms of Curry types. These are taken from the standard `flatcurry` [Han17c] and `flatcurry-annotated` [Han17b] packages and implement the syntax defined in figure 2.2 and are used extensively throughout the verification framework. For brevity, we omit the **`Eq`**, **`Ord`**, **`Read`** and **`Show`** instances, which are derived for all types using the standard **`deriving`** clause.

A.1 Untyped FlatCurry

```
-- A FlatCurry program/module.
data Prog = Prog String [String] [TypeDecl] [FuncDecl] [OpDecl]

-- A qualified name (module name, unqualified name).
type QName = (String, String)
-- The public/private visibility of an entity.
data Visibility = Public | Private

-- An index representing a type variable.
type TVarIndex = Int
-- A kinded type variable.
type TVarWithKind = (TVarIndex, Kind)

-- A type declaration.
data TypeDecl
  = Type    QName Visibility [TVarWithKind] [ConsDecl]  -- A `data` type
  | TypeSyn QName Visibility [TVarWithKind] TypeExpr    -- A type synonym
  | TypeNew QName Visibility [TVarWithKind] NewConsDecl -- A `newtype`

-- A data constructor.
data ConsDecl = Cons QName Int Visibility [TypeExpr]
-- A newtype constructor.
data NewConsDecl = NewCons QName Visibility TypeExpr

-- A type expression.
data TypeExpr
  = TVar TVarIndex                      -- A type variable
  | FuncType TypeExpr TypeExpr          -- A function type of the form `t1 -> t2`
  | TCons QName [TypeExpr]              -- Type constructor application
  | ForallType  [TVarWithKind] TypeExpr  -- A forall type
```

```
-- A type kind.
data Kind = KStar | KArrow Kind Kind
-- An operator declaration.
data OpDecl = Op QName Fixity Int
-- An operator fixity.
data Fixity = InfixOp | InfixlOp | InfixrOp
-- An index representing a variable.
type VarIndex = Int
-- A function arity.
type Arity = Int

-- A function declaration.
data FuncDecl = Func QName Arity Visibility TypeExpr Rule

-- A function implementation.
data Rule
  = Rule [VarIndex] Expr            -- A list of formal params and a body.
  | External String                 -- An "external" tag.

-- Classifies case expressions.
data CaseType
  = Rigid                           -- Uses Haskell-like match semantics
  | Flex                            -- Uses Curry's semantics for overlapping patterns

-- Classifies combinations.
data CombType
  = FuncCall                        -- Call to a function with all args provided
  | ConsCall                        -- Call to a constructor with all args provided
  | FuncPartCall Arity
  | ConsPartCall Arity

-- An expression.
data Expr
  = Var VarIndex                    -- A variable (represented by unique index)
  | Lit Literal                     -- A literal (int/float/char constant)
  | Comb CombType QName [Expr]      -- Application of the form `f e1 ... en`
  | Let [(VarIndex, Expr)] Expr     -- Introduction of (recursive) local bindings
  | Free [VarIndex] Expr            -- Introduction of free local vars
  | Or Expr Expr                    -- Disjunction of two expressions (choice)
  | Case CaseType Expr [BranchExpr] -- Case distinction (rigid or flex)
  | Typed Expr TypeExpr             -- Explicitly typed expression

-- A branch in a case expression.
data BranchExpr = Branch Pattern Expr

-- A pattern a in case expression.
data Pattern = Pattern QName [VarIndex] | LPattern Literal
```

```
-- A literal in an expression or branch.
data Literal
  = Intc   Int
  | Floatc Float
  | Charc  Char
```

A.2 Annotated FlatCurry

```
-- An annotated FlatCurry program.
data AProg a = AProg String [String] [TypeDecl] [AFuncDecl a] [OpDecl]

-- An annotated function declaration.
data AFuncDecl a = AFunc QName Arity Visibility TypeExpr (ARule a)

-- An annotated function rule.
data ARule a
  = ARule     a [(VarIndex, a)] (AExpr a)
  | AExternal a String

-- An annotated expression.
data AExpr a
  = AVar   a VarIndex
  | ALit   a Literal
  | AComb  a CombType (QName, a) [AExpr a]
  | ALet   a [((VarIndex, a), AExpr a)] (AExpr a)
  | AFree  a [(VarIndex, a)] (AExpr a)
  | AOr    a (AExpr a) (AExpr a)
  | ACase  a CaseType (AExpr a) [ABranchExpr a]
  | ATyped a (AExpr a) TypeExpr

-- An annotated case branch.
data ABranchExpr a
  = ABranch (APattern a) (AExpr a)

-- An annotated pattern.
data APattern a
  = APattern  a (QName, a) [(VarIndex, a)] -- A constructor pattern
  | ALPattern a Literal                    -- A literal pattern
```

Since annotated FlatCurry is primarily used with type annotations, we define the following synonyms:

```
type TAProg       = AProg       TypeExpr
type TAFuncDecl   = AFuncDecl   TypeExpr
type TARule       = ARule       TypeExpr
type TAExpr       = AExpr       TypeExpr
type TABranchExpr = ABranchExpr TypeExpr
type TAPattern    = APattern    TypeExpr
```

Appendix B

Framework Configuration

This appendix contains a comprehensive overview over the client-customizable configuration of the verification framework, along with its defaults and the referenced logging API.

B.1 Options

The `VOptions` type is the central configuration facility that provides the name of the verification, a list of modules to verify and a number of other flags to the framework. As described in section 3.4.2, this type is passed to one of the framework's entry points. It is defined as follows:

```
data VOptions a = VOptions
  { voName             :: String                -- The name of the verification. Used as a cache key.
  , voModules          :: [String]              -- The source paths to the Curry modules to analyze
                                                -- (with or without .curry suffix)
  , voLog              :: VMessage -> IO ()     -- The logging backend to use. By default log
                                                -- messages are simply discarded.
  , voPpFuncInfo       :: Maybe (a -> String)   -- Pretty-prints a function info/verification value,
                                                -- for debug logs.
  , voCacheEnabled     :: Bool                  -- Whether caching is enabled.
  , voCacheKeys        :: [String]              -- Keys to use for caching (in addition to the name).
                                                -- Any of these changing will invalidate the cache.
  , voCacheRootDir     :: FilePath              -- The root directory for all verification caches.
  , voCacheSubdir      :: FilePath              -- The directory within the verification root dir to
                                                -- place all caches in, mostly for versioning.
  , voMaxIterations    :: Int                   -- The maximum number of iterations
  , voPreludePrimOps   :: [(String, String)]    -- Primitive ops from the Prelude and their SMT names
  , voUnaryPrimOps     :: [(String, String)]    -- Unary primitive operations to use during Simplify
  , voBinaryPrimOps    :: [(String, String)]    -- Binary primitive operations to use during Simplify
  , voWriteProgs       :: Bool                  -- Write the transformed progs in the chosen format
  , voWriteUntypedProgs :: Bool                 -- Write the transformed progs as untyped FlatCurry
                                                -- Only has an effect using type-annotated FlatCurry.
  , voSkipPrelude      :: Bool                  -- Skip verification of the Prelude. Can make
                                                -- verifications more performant if they don't need
                                                -- verification infos from the Prelude.
  , voEnforceNF        :: Bool                  -- Evaluate program state (i.e. programs and
                                                -- verification values) strictly/to normal form
                                                -- (i.e. deeply) in every iteration. This may save
                                                -- memory in some cases.
  }
```

For convenience, we also define a helper function for constructing a default configuration:

```
defaultVOptions :: String -> IO (VOptions a)
defaultVOptions name = do
  home <- getHomeDirectory
  let currySystemSubdir = joinPath (tail (splitDirectories currySubdir))
  return VOptions
    { voName                = name
    , voModules             = []
    , voLog                 = noLog
    , voPpFuncInfo          = Nothing
    , voCacheEnabled        = False -- Experimental for now
    , voCacheKeys           = []
    , voCacheRootDir        = home </> ".curry_verification_cache"
    , voCacheSubdir         = "v1" </> currySystemSubdir
    , voMaxIterations       = 16
    , voPreludePrimOps      = []
    , voUnaryPrimOps        = []
    , voBinaryPrimOps       = []
    , voWriteProgs          = False
    , voWriteUntypedProgs = False
    , voSkipPrelude         = False
    , voEnforceNF           = False
    }
```

B.2 Logging API

The options listed above include a log handler taking a `VMessage`. This is defined as follows:

```
data VMessage = VMessage
  { vmLevel :: VLevel
  , vmText  :: String
  }
  deriving (Show, Eq, Ord)

data VLevel = VLevelAll | VLevelDebug | VLevelInfo | VLevelWarn | VLevelError | VLevelNone
  deriving (Show, Eq, Ord)
```

We also provide some standard log handlers, along with a `withVLevel` combinator:

```
noLog :: Applicative m => VMessage -> m ()
noLog _ = pure ()

printLog :: VMessage -> IO ()
printLog = putStrLn . pPrint . ppVMessage

withVLevel :: Applicative m => VLevel -> (VMessage -> m ()) -> VMessage -> m ()
withVLevel l f m = when (vmLevel m >= l) $ f m
```

Appendix C

List of Modules

Below is a comprehensive list of the framework's modules, summarizing each module briefly together with its most relevant exports. The list serves both as documentation for the code-level implementation of the framework and as a reference for how the design and architecture described earlier map to concrete modules.

C.1 Public Modules

The framework includes the following modules for public use[1]:

▻ `Verification.Env` contains the environments exposing read-only, contextual information to the client-provided verification. There are three kinds of environments, providing varying levels of context to the verification:

  ▻ `VBaseEnv`, the base environment, hosting options and mapping functions between FlatCurry programs and function declarations, along with all global state, including read programs, tracked functions and verification results.

  ▻ `VProgEnv`, the program environment, which is a superset of the base environment. In addition to the base environment it tracks a currently visited program.

  ▻ `VFuncEnv`, the function environment, which is a superset of the program environment, In addition to the program environment it tracks a currently visited function.

▻ `Verification.FlatCurry` hosts utility functions and type synonyms for untyped FlatCurry

▻ `Verification.FlatCurry.Annotated` hosts utilities for type-annotated FlatCurry

▻ `Verification.Handlers` provides the `VProgHandlers` type, an aggregation of utility functions on FlatCurry programs, e.g. for extracting the list of all functions from a program, for fetching the name or dependencies of a function, for reading and writing FlatCurry programs to disk etc. The primary purpose of these mappings is to abstract over typed and untyped FlatCurry in a uniform way as many of these functions delegate to the corresponding FlatCurry goodies at run time.

▻ `Verification.Info` provides the `VProgInfo` type, which is a map of all function infos for a program.

▻ `Verification.Log` provides a logging abstraction used by the framework, including logging levels and some standard logging implementations, e.g. for outputting messages to the console or ignoring them.

[1]Note that some of these modules are internally divided into finer-grained modules, such as `Verification.Env.Base`, which are reexported by their parents, in this case `Verification.Env`. While verification clients are allowed to import these modules directly too, it is recommended to always using the parent modules, both for brevity and for improved API stability.

- `Verification.Monad` provides the `VM` monad, defined as `type VM = ExceptT String IO`
  This monad is used in the functions implemented by the client and provides the capability to perform I/O and throw errors purely, i.e. without resorting to run-time errors.
- `Verification.Options` provides `VOptions`, the core set of flags and configurations that the client passes to the framework via the chosen entry point. This includes e.g. the name of the verification, which is used for caching purposes, a list of modules to be verified and a logging backend.
- `Verification.Run` provides the framework's entry points, specifically `runUntypedVerification` and `runTypeAnnotatedVerification`, for untyped and type-annotated FlatCurry, respectively.
- `Verification.State` provides the `VProgState` type, which stores a FlatCurry program and its associated verification results, i.e. a `VProgInfo`, along with the higher-level `VState` type, which stores a map of module names to `VProgState`.
- `Verification.Types` provides the core `Verification` type, hosting the functions implemented by the client. Like `VOptions`, this type is passed to the entry point.
- `Verification.Update` provides types that clients use to communicate back changes to the framework, essentially taking on the opposite role to environments. Like with the environments there are different types of updates, depending on the context, though it should be noted that each of these update types has an empty default value that corresponds to the no-op update:
  - `VProgUpdate` encapsulates an update of a FlatCurry program prior to verification, primarily for preprocessing. This type optionally contains an updated program.
  - `VFuncUpdate` encapsulates an update of a FlatCurry function during verification. This type can both communicate a mutation of the program by including an updated version of the function, e.g. adding dynamic checks, and even a list of new functions, as well as updates to the verification results, by including an updated info for the current function and other functions in the program.

C.2 Internal Modules

The implementation also includes three more modules that are considered implementation details of the framework, hosting internal logic, and are not intended for use by clients:

- `Verification.Internal.Cache` includes the cache schema, i.e. data types that the frameworks writes to the cache directory, which by default is under `~/.curry_verification_cache`.
- `Verification.Internal.Monad` implements the internally used `VIM` monad. This monad is a superset of the external `VM` monad that additionally tracks the client-provided options, the handlers abstracting over FlatCurry functions, along with all global state, including read programs, tracked functions and computed verification results. Additionally, the module also hosts some helper functions that are implemented in terms of this monad.
- `Verification.Internal.Run` implements the `runVerification` function, performing the core logic driving the verification and invoking the client-provided verification functions at the appropriate stages. This function generically handles both untyped and type-annotated programs and is called by the public entry points, i.e. `runUntypedVerification` and `runTypeAnnotatedVerification`.
- `Verification.Internal.Utils` provides a set of utility functions used internally by the framework. These do not depend on any other project modules.

Appendix D

SMT Translation Examples

In this appendix we provide two full examples of FlatCurry functions translated to SMT for the construction of size-change graphs in the context of the termination checker, as described in section 5.2.5.

D.1 Factorial

This example showcases the factorial function, defined as follows:

```
fac'pre :: Int -> Bool
fac'pre n = n >= 0

fac :: Int -> Int
fac n = if n == 0 then 1
                  else n * fac (n - 1)
```

The implementation of `fac` is translated verbatim to the following SMT program:

```
; Arguments
(declare-const x1 Int)

; Preconditions
(assert (>= x1 0))

; case Prelude._impl#==#Prelude.Eq#Prelude.Int# v1 0 of
(declare-const tmp0 Bool)
(assert (= tmp0 (= x1 0)))
; Prelude.True ->
(push)
  (assert (= tmp0 true))
(pop)

; Prelude.False ->
(push)
  (assert (= tmp0 false))
  (push)
    ; call to Prelude._impl#*#Prelude.Num#Prelude.Int#
    (push)
      ; call to Factorial.fac
      (echo
      "Call {cSrcFunc = (""Factorial"",""fac""), cDstFunc = (""Factorial"",""fac"")}")
      (echo "Edge {eSrcParam = 0, eDstParam = 0}")
      (push)
        (assert (>= (abs (- x1 1)) (abs x1)))
        (check-sat)
      (pop)
      (push)
        (assert (not (= (- x1 1) x1)))
        (check-sat)
      (pop)
    (pop)
    (echo
    "Call {cSrcFunc = (""Factorial"",""fac""),
           cDstFunc = (""Prelude"",""_impl#*#Prelude.Num#Prelude.Int#"")}")
    (echo "Edge {eSrcParam = 0, eDstParam = 0}")
    (push)
      (assert (>= (abs x1) (abs x1)))
      (check-sat)
    (pop)
    (push)
      (assert (not (= x1 x1)))
      (check-sat)
    (pop)
  (pop)
(pop)
```

Passing this to Z3 produces the following output:

```
Call {cSrcFunc = ("Factorial","fac"), cDstFunc = ("Factorial","fac")}
Edge {eSrcParam = 0, eDstParam = 0}
unsat
sat
Call {cSrcFunc = ("Factorial","fac"), cDstFunc = ("Prelude","_impl#*#Prelude.Num#Prelude.Int#")}
Edge {eSrcParam = 0, eDstParam = 0}
sat
unsat
```

D.2 Fibonacci

Consider the Fibonacci function, defined as follows:

```
fib'pre :: Int -> Bool
fib'pre n = n >= 0

fib :: Int -> Int
fib n = case n of
  0 -> 0
  1 -> 1
  _ -> fib (n - 2) + fib (n - 1)
```

Here, the `fib` function translates to the following SMT program:

```
; Arguments
(declare-const x1 Int)

; Preconditions
(assert (>= x1 0))

; case Prelude._impl#==#Prelude.Eq#Prelude.Int# v1 0 of
(declare-const tmp0 Bool)
(assert (= tmp0 (= x1 0)))
; Prelude.True ->
(push)
  (assert (= tmp0 true))
(pop)

; Prelude.False ->
(push)
  (assert (= tmp0 false))
  (push)
    ; case Prelude._impl#==#Prelude.Eq#Prelude.Int# v1 1 of
    (declare-const tmp1 Bool)
    (assert (= tmp1 (= x1 1)))
    ; Prelude.True ->
    (push)
      (assert (= tmp1 true))
    (pop)

    ; Prelude.False ->
    (push)
      (assert (= tmp1 false))
      (push)
        ; call to Prelude._impl#+#Prelude.Num#Prelude.Int#
        (push)
          ; call to Fibonacci.fib
          (echo
          "Call {cSrcFunc = (""Fibonacci"",""fib""), cDstFunc = (""Fibonacci"",""fib"")}")
          (echo "Edge {eSrcParam = 0, eDstParam = 0}")
          (push)
            (assert (>= (abs (- x1 2)) (abs x1)))
            (check-sat)
          (pop)
          (push)
            (assert (not (= (- x1 2) x1)))
            (check-sat)
          (pop)
        (pop)
        (push)
          ; call to Fibonacci.fib
          (echo
          "Call {cSrcFunc = (""Fibonacci"",""fib""), cDstFunc = (""Fibonacci"",""fib"")}")
          (echo "Edge {eSrcParam = 0, eDstParam = 0}")
          (push)
            (assert (>= (abs (- x1 1)) (abs x1)))
            (check-sat)
          (pop)
          (push)
            (assert (not (= (- x1 1) x1)))
            (check-sat)
          (pop)
        (pop)
        (echo
        "Call {cSrcFunc = (""Fibonacci"",""fib""),
               cDstFunc = (""Prelude"",""_impl#+#Prelude.Num#Prelude.Int#"")}")
      (pop)
    (pop)

  (pop)
(pop)
```

Passing this to Z3 produces the following output:

```
Call {cSrcFunc = ("Fibonacci","fib"), cDstFunc = ("Fibonacci","fib")}
Edge {eSrcParam = 0, eDstParam = 0}
unsat
sat
Call {cSrcFunc = ("Fibonacci","fib"), cDstFunc = ("Fibonacci","fib")}
Edge {eSrcParam = 0, eDstParam = 0}
unsat
sat
Call {cSrcFunc = ("Fibonacci","fib"), cDstFunc = ("Prelude","_impl#+#Prelude.Num#Prelude.Int#")}
```

Appendix E

Parser Combinators

A side product of the SMT translation developed for the termination checker is a small, self-contained library module for combinatory parsing. The module is heavily inspired by Parsec, one of the most well-known monadic parser combinator libraries for Haskell [HM96; LM01] and is based on a simplified version of these combinators [Pau22].

E.1 Overview

Fundamentally, our parser combinators are centered around `Parser`, a monadic parser combinator type that parses an `s` to an `a`, both of which are intentionally kept as generic as possible:

```haskell
newtype Parser s a = Parser { runParser :: s -> Either String (a, s) }
```

This means the type is generic enough to handle both character-level parsing when `s` is a `String` and e.g. line-level parsing by setting `s` to `[String]`. The signature of `runParser` takes an `s` and returns an `(a, s)` representing both the parsed result and an unparsed remainder, wrapped in an `Either String` to provide a way of throwing errors. As mentioned above, `Parser` forms a monad, whose instance can be defined as follows:

```haskell
instance Monad (Parser s) where
  return x = Parser $ \s -> Right (x, s)
  p >>= f = Parser $ \s -> do
    (x, s') <- runParser p s
    runParser (f x) s'
```

The `Functor` and `Applicative` can be directly derived from this instance. We also provide an `Alternative` instance to represent choice, i.e. a parser that first tries to parse the left-hand side and, if that fails, the right-hand side:

```haskell
instance Alternative (Parser s) where
  empty     = Parser . const $ Left "Empty parser"
  p1 <|> p2 = Parser $ \s -> case runParser p1 s of
    Right (x, s') -> Right (x, s')
    Left _        -> runParser p2 s
```

It should be noted that this implements choice with backtracking, as the invocation of `runParser p2` takes the original string `s`. This is usually avoided in popular Parsec-style libraries as naive usage may quickly result in exponential run time due to effectively implementing infinite lookahead. Other libraries require the left-hand side to not consume any input and then provide a `try` combinator to explicitly opt in to backtracking, but since we directly control the application of these combinators, we opt for this slightly simpler implementation.

The most primitive combinator that we provide is `satisfy`, which effectively matches the next atom of the input against a predicate, e.g. a character in a `Parser String` or a line in a `Parser [String]`:

```
satisfy :: (c -> Bool) -> Parser [c] c
satisfy f = Parser $ \cs -> case cs of
  c:cs' | f c       -> Right (c, cs')
        | otherwise -> Left "Not satisfied"
  []                -> Left "EOF too early in satisfy"
```

Using this, we can easily define a combinator that matches the atom directly:

```
single :: Eq c => c -> Parser [c] c
single c = satisfy (== c)
```

We also implement a combinator that matches multiple atoms, e.g. a `String` in a `Parser String`:

```
literal :: (Show c, Eq c) => [c] -> Parser [c] [c]
literal cs = Parser $ match cs
  where match cs1 cs2 = case (cs1, cs2) of
          ([], _)                       -> Right (cs, cs2)
          (c1:cs1', c2:cs2') | c1 == c2  -> match cs1' cs2'
                             | otherwise -> Left $ "Could not parse " ++ show cs ++ ": "
                                                ++ show c1 ++ " != " ++ show c2
          _                             -> Left $ "EOF while parsing " ++ show cs
```

We also define higher-order parser combinators, including one that parses one or more instances of a given parser:

```
many1 :: Parser s a -> Parser s [a]
many1 p = do
  x  <- p
  xs <- many1 p <|> return []
  return (x : xs)
```

Using `many1` and the choice operator `(<|>)`, we can also define a combinator that parses zero or more instances:

```
many0 :: Parser s a -> Parser s [a]
many0 p = many1 p <|> return []
```

Occasionally it can be convenient to explicitly lookahead without consuming any input. This can be done too via a combinator[1]:

```
lookAhead :: Parser s a -> Parser s a
lookAhead p = Parser $ \s -> (\(x, _) -> (x, s)) <$> runParser p s
```

Another useful combinator is `subparse`, which effectively lifts a combinator to one that operates on lists, e.g. to take a character-level `Parser String` to a line-level `Parser [String]`:

```
subparse :: Parser c a -> Parser [c] a
subparse p = Parser $ \cs -> case cs of
  c:cs' -> (\(x, _) -> (x, cs')) <$> runParser p c
  []    -> Left "EOF too early in subparse"
```

[1]The idea behind this combinator is similar to the `try` combinator that other Parsec-style libraries provide for backtracking which we mentioned earlier. Unlike `try`, `lookAhead` does not consume any input *when the input matches* whereas `try` is usually defined to not consume input *when the input does not match*, i.e. when an error occurs. This is a use case which we do not support, as the left side of the `Either` is `String` and does not carry a remaining input s. Since we backtrack by default, this is not needed either, however.

Note that we discard the rest of the subparse, e.g. the rest of line if operating on lines of strings here, so it should be used carefully to avoid discarding any input by accident.

For convenience, we also define a combinator that wraps the `read` function from the `Read` type class, to simplify deserialization:

```haskell
readp :: Read a => Parser String a
readp = Parser $ \s ->
  case reads s of
    [(x, s')] -> Right (x, s')
    _         -> Left "Could not parse with Read"
```

Finally, we also provide a `(<?>)` combinator to attach a custom error message, aiding both debugging and error reporting:

```haskell
(<?>) :: Parser s a -> String -> Parser s a
p <?> e = Parser $ \s -> case runParser p s of
  Right x -> Right x
  Left e' -> Left $ e' ++ ": " ++ e
```

E.2 Example

The canonical example where we apply these combinators is the parser for the output of the SMT-based size change graph construction of the termination checker as described in section 5.2.5 and e.g. shown in appendix D. This parser is implemented as follows:

```haskell
graph :: Parser [String] Graph
graph = do
  c  <- subparse call <?> "Expecting call"
  es <- Map.fromList . catMaybes <$> many0 labeledEdge
  return $ Graph c es

call :: Parser String Call
call = lookAhead (literal "Call") >> readp

labeledEdge :: Parser [String] (Maybe (Edge, SizeChange))
labeledEdge = do
  e  <- subparse edge <?> "Expecting edge"
  sc <- sizeChange    <?> "Expecting size change"
  return $ (,) e <$> sc

edge :: Parser String Edge
edge = lookAhead (literal "Edge") >> readp

sizeChange :: Parser [String] (Maybe SizeChange)
sizeChange = do
  smaller <- not <$> subparse sat
  equal   <- not <$> subparse sat
  return $ if smaller then Just SmallerSize else if equal then Just EqualSize else Nothing
```

```
sat :: Parser String Bool
sat = (const True <$> literal "sat") <|> (const False <$> literal "unsat")
```

This parser is then applied as follows to produce a list of graphs from a raw string:

```
parseZ3Output :: String -> VM [Graph]
parseZ3Output raw =
  case runParser (many0 graph) (lines raw) of
    Right (gs, _) -> return gs
    Left e        -> throwVM $ "Could not parse Z3 output: " ++ e
```

Appendix F

Source Code

The source code of the verification framework, along with the termination checker and other examples, is hosted on GitHub and can be found at `https://github.com/fwcd/curry-verification`. The port of the contract prover can be found at `https://github.com/fwcd/curry-contract-prover` under the `verification-framework` branch. Documentation on how to build and run them can be found in the respective `README.md` files.