Project

DEPENDENT TYPES FOR REAL-TIME CONSTRAINTS

by

WILLIAM BLAIR
B.A., Boston University, 2012

Submitted in partial fulfillment of the
requirements for the degree of
Master of Science
2014
DEPENDENT TYPES FOR REAL-TIME CONSTRAINTS
WILLIAM BLAIR

ABSTRACT

Synchronous programming languages emerged in the 1980s as tools to implement reactive systems. These systems interact with events from their physical environment and often must do so under strict time constraints. These languages’ declarative syntax and temporal semantics raise the level abstraction for real time development. As a result, they gained use in industry and are still used today.

In this thesis, we use ATS, a language with an advanced type system, to support real-time primitives in an experimental synchronous language called Prelude. We show that these primitives and their verification requirements may be expressed using dependent types in ATS, and then modify the Prelude compiler to produce ATS code for type checking. This modified Prelude compiler thus discharges part of its proof obligations to ATS. Whereas ATS is typically used as a general purpose programming language, we argue here that it may be used to support verification features in more simply typed languages as well.

In order to support this, we outline a key short coming in the current ATS type checker that restricts dependent types to capture invariants involving linear integer arithmetic. We replace the constraint solver with an external tool that utilizes the Z3 SMT Solver. This drastically simplifies the constraint solver and also allows us to verify more program properties using dependent types.
## Contents

1 Introduction  

2 Synchronous Languages  
   2.1 Expressing Multi-Periodic Systems  
   2.2 The Prelude Programming Language  
   2.3 Clocks in Prelude  

3 Overture: Expressing Prelude in ATS  
   3.1 Dependent Types in ATS  
   3.2 Types for Prelude Operators  
   3.3 ATS as a Verification Back End  
   3.4 Possible Extension  

4 Enriching Dependent Types in ATS  
   4.1 Requirements for a Constraint Solver  
   4.2 Adding New Sorts to ATS Statics  

5 Related Work  

6 Conclusions  

Bibliography
Chapter 1

Introduction

Building software that must work with a high level of confidence is an undeniably difficult process. Yet, people are still able to construct reliable systems that operate under strict temporal requirements. The language of choice in these systems is still low level languages such as C/C++ and Ada. Recently companies have provided model based approaches for real time software. Model based methods take requirements and descriptions of a system and generate C code that may be run with high correctness guarantees within a real time operating system.

System implementers always desire high level tools that deliver a reliable product in a fast development cycle. Constructing a reactive system with deterministic behavior using C inspires neither reliability nor fast development. The standard techniques a C programmer may to use to implement communication between processes such as locks, semaphores, and monitors, are of little use to real time systems because of the potential risks of deadlock. These risks introduce non-determinism into the system. For mission critical processes is simply unacceptable.

Synchronous programming languages were first developed in the 1980s and introduce the notion of a global clock in the context of a program. Languages similar to LUSTRE[6] have since moved into industry where they are employed in critical control systems from aircraft to nuclear reactors. In a synchronous paradigm, an engineer describes a system of communicating tasks by describing how data flows between processes. From this description, a compiler generates code that will either run on the bare metal or in a real time operating system. In either case, the produced program conforms to the specification set
forth by the original design. This model based design approach drastically reduces the time
needed to build reliable code. Yet, some tasks still recall the tedium of manual coding. Im-
plementing a multi-periodic system in a language where all flows occur in sync with a base
clock is difficult and error prone. Suddenly, time becomes a real concern as the designer
must manually adjust flows to meet specific timing requirements. What’s more, the differ-
ent timing requirements are unbeknown est to the compiler. The programmer must then
rely on simulation and testing to check correctness. There is nothing wrong with testing
and simulating a system, but if the compiler can make use of important information like
real time characteristics for static analysis, then it is important to do so.

Julien Forget addressed this issue in his PhD thesis where he outlined an extension
of synchronous languages like LUSTRE called Prelude to support multi-periodic systems.
These new additions allow a programmer to express flows that occur with different clocks.
This contrasts earlier synchronous languages that abstract away real time properties in
favor of a single global clock. This flexibility obligates the compiler to solve constraints
derived from a Prelude program so that communication between tasks is always synchro-
nized. In this thesis, we use ATS as an intermediate verification language for handling
this phase of type checking. In doing so, we demonstrate ATS’ type system facilitating
advanced features in other programming languages using a combination of abstract and
dependent types.

Before we can use ATS to verify Prelude code, we need to address the limited types
available in the ATS statics. Real time tasks are described by the periods and phase offsets
at which they run, which may be expressed using rational numbers. In order to capture
constraints involving these parameters, our type system must understand both integer and
rational arithmetic. So far, rational numbers have been absent from ATS statics which the
domain of ATS concerned with constraint solving. In order to remedy this issue, we outline
how one can extend the statics in ATS with arbitrary sorts, and defer proof obligations
to an external solver. In this case, we develop a constraint solver on top of the Z3 SMT
solver. With this more advanced solver we can enforce invariants over real time clocks and
give Prelude operators precise ATS types. With this system we may present Overture, a modified Prelude compiler that transforms Prelude programs into ATS for type checking.
Chapter 2

Synchronous Languages

Synchronous programming languages provide a high level of abstraction for implementing reactive systems. A reactive system is one that must run continuously while interacting with its external environment. In this thesis, we will only describe the syntax and semantics of declarative languages like LUSTRE, as they are the basis for Prelude. A more complete description of both Imperative and Declarative synchronous languages may be found in either Forget’s thesis [5] or a survey on the use and evolution of synchronous languages in industry [1].

In the declarative synchronous languages, the real time behavior of code is abstracted away in favor of a logical time scale. The purpose of programming in these languages is primary to describe the flow of data through a program’s execution. Imported nodes are available The primary building block in a program is a flow.

Flows are streams of values of some primitive type that can be thought as being produced instantaneously in step with some logical clock. Of course, it does take time for a value to be generated, but this is just a part of the abstraction. As an example, suppose our base clock is a fixed value $C$ and consider the flow $F$. Let $F_i$ be the $i^{th}$ value in $F$. From the definition of flows, we know that at date $t_i$ the value of $F$ is $F_i$.

This is a useful because it allows the programmer to describe the communication of data in his or her program without specifying explicitly when each value is produced. Instead, the compiler will translate the synchronous description into a traditional program that computes the instance of all flows in a single step. The order of this evaluation is given from the synchronous program so all data dependencies between nodes is known.
beforehand.

In this paradigm, the tedious programming task of manually writing communication between concurrent tasks in a language like C is lifted to a much higher level of abstraction. This helps eliminate many common errors that could have been made if the program was written manually and leads to software that may be considered “Correct by Construction”. Of course, in order for the resulting software to be correct, it must be verified under the Synchronous Hypothesis, which we hinted at in our definition of flows.

When a synchronous program is compiled, all nodes and flows are transformed into a single step function that is called repeatedly. The time between steps is the period of the system, and the Synchronous Hypothesis states that all flows must be evaluated at before the end of each step. It is up to the implementor to determine whether or not a program meets this requirement.

2.1 Expressing Multi-Periodic Systems

In typical synchronous programming languages, all flows occur in step with a global base clock. If communicating tasks occur with different time scales, then you must manually over or under sample flows to meet your desired behavior. We refer to these systems as multi-periodic. A multi-periodic system is comprised of mono-periodic tasks that occur at different time scales. As an example, consider a system between two processes that periodically communicate with one another, but one executes 10 times faster than the other. This example comes from Forget’s thesis and is featured in the motivation for the Prelude project.

In order for these two tasks to communicate, you need to under-sample the flow coming from the faster task into the slower task. This produces a flow with the same clock as the slower process so that they are synchronous. The problem with this method in a language like LUSTRE is that these under and over sampling operations are absent from LUSTRE’s semantics. Therefore, it can be difficult for the programmer to infer from
code that one task, in fact, occurs 10 times less frequently than another. This lack of clarity in the programmer’s mind can lead to timing bugs in the final program. What’s more, the compiler has no way to check that this inconsistency exists in the program a priori, and so finding it will be deferred to testing and simulation. Abstracting real time from development affords the synchronous paradigm a pleasant level of abstraction. Applications that are multi-periodic with many different time scales can be difficult to implement in this setting, but introducing primitives to express real-time behavior in the language could simplify development.

2.2 The Prelude Programming Language

Prelude[5] addresses the timing issues found in synchronous languages by tagging each flow with a clock and enforcing that flows between communicating tasks are synchronous (i.e. they have the same clock). This is done by checking that each clock transformation’s pre-conditions are satisfied under a “clock-calculus” proposed by Forget.

The process of checking each clock for validity is a form of constraint solving. For this purpose, Forget designed and implemented a custom decision algorithm for checking programs for conformance to the language’s clock calculus. In contrast to other Synchronous languages, the Prelude compiler is aware of important real time characteristics of flows and the nodes that produce and consume them. This helps it both catch timing errors and facilitate assembling mono-synchronous tasks into a multi-periodic system.

2.3 Clocks in Prelude

Prelude’s main departure from contemporary synchronous languages is the ability to describe real time behavior of flows. There is also support for input and output of the system, which are simply flows called sensors and actuators, respectively.

Every flow $F$ has an associated clock $\alpha$ represented as a pair $(n, p)$ such that $n \in \mathbb{N}^+$ and $p \in \mathbb{Q}^+$. The value $p$ denotes the period of $\alpha$ and $p$ is phase offset expressed as $\pi(\alpha)$
and $\phi(\alpha)$, respectively. The period of a flow $F$ is the amount of time between occurrences of values produced by $F$ and the phase offset is the initial delay of the flow. Let $t_0$ be the date of the first element of a flow and it may be expressed as the following.

**Definition 2.3.1**

\[ t_0 = \pi(\alpha) \cdot \phi(\alpha) \]

In Prelude’s clock calculus, all dates (points in time in the program) are restricted to be integers in order to simplify the schedulability analysis. Therefore, we have to ensure that the date $t_0$ is an integer for all flows. Therefore, the clock $(n, p)$ is valid if and only if $n \in \mathbb{N}^+$ and $n \cdot p \in \mathbb{N}^+$.

Under the normal synchronous paradigm, all flows produce some value during each tick of the logical clock. In Prelude, we no longer have a notion of a global time scale. Instead, every flow now has its own clock which we can lengthen or shorten as needed. By modifying a clock we are changing the rate at which values are produced in a program. If the values contained in a flow is undefined for some arbitrary dates, we cannot adjust its real time behavior and expect deterministic results. As such, it is useful to define a class of clocks that are strictly periodic. A strictly periodic clock is subject to the constraint that a value must be produced after each period. Put more formally, a clock $C$ is strictly periodic if and only if

\[ \exists n \in \mathbb{Q}^+, \forall i \in \mathbb{N} t_{i+1} - t_i = n \]

Where $n$ is $\pi(C)$ and $t_i$ is the date of the $i^{th}$ value in a flow with clock $C$. Prelude provides a set of operators to transform strictly periodic clock. The primary purpose of these operators is to facilitate communication between tasks whose rate of execution differ.

**Definition 2.3.2** (Multiplication (over-sampling))

\[ \forall (n, p) \in \mathbb{N} \times \mathbb{Q}^+, \; k \in \mathbb{N}^+ \; (n, p) \cdot k \rightarrow \left( \frac{n}{k}, p \cdot k \right) \]

**Definition 2.3.3** (Division (under-sampling))

\[ \forall (n, p) \in \mathbb{N} \times \mathbb{Q}^+, \ k \in \mathbb{N}^+, \ (n, p)/k \rightarrow (n \cdot k, \frac{p}{k}) \]

**Definition 2.3.4 (Delay)**

\[ \forall (n, p) \in \mathbb{N} \times \mathbb{Q}^+, \ k \in \mathbb{Q}, \ (n, p) \rightarrow k \rightarrow (n, p+k) \]

Multiplication and division of clocks allow us to under-sample and over-sample flows, respectively. The delay operator allows us to adjust the In all these definitions both the clocks given to the operators and their result must be valid clocks. Using these transformations in a Prelude program obligates the compiler to ensure that all clocks are valid and strictly periodic. Each time a clock is transformed a new constraint is made to determine whether the clock is valid. The process of type checking traverses the entire abstract syntax tree of a Prelude program.

Consider the example in the previous section where we have two communicating tasks, one of which occurs 10 times faster than the other. To make the system more concrete, let’s consider a piece of code samples a sensor ten times per second. A slower controller tasks runs only once per second and queries the sampler for information depending on some external input it receives. The controller then responds through an actuator that also has a rate of one second. We can use the operators we defined above to describe this communication in the following example snippet of Prelude.

```prelude
node sample (i) returns (o)
node controller (i, data) returns (cmd, rep)

node main (i: rate (1, 0)) returns (o)

    var command, response;

    let
        (o, command) = controller (i, (0 fby response) *^ 10)
        response = sample (command /^ 10)

    tel
```

One unfamiliar operator presented above is `fby`, which is just syntactic sugar for
0 :: response ⇝ 1. The purpose of fby is to break the circular dependency between the command and response flows. We accomplish this by setting the first value of rep response flow to be zero, followed by the next value of response. This is given by the first value of sample. This breaks the circular dependency while also preserving the clock of response.

This “clock calculus” also supports the standard operators from synchronous languages such as LUSTRE. These operators do not modify the clock of the flow with exception of the when operator. Recall that the purpose of when is to under-sample a flow based on the values of a boolean flow. By using when we can activate values of a flow only when the values from another flow evaluate to true. In Prelude, both flows are required to have the same clock and be strictly periodic. Note that the result of when is obviously not strictly periodic; the dates between values in the produced flow may differ arbitrarily.

If we want to make flows conditional on other boolean flows, in order to address this, we can use the merge operator. This operator yields a strictly periodic flow because it takes a boolean flow b and two other flows on and off. The flow given by merge(b, on, off) has value on_i at date t_i if b_i = true and off_i otherwise.
Chapter 3

Overture: Expressing Prelude in ATS

In the previous chapter, we described the Prelude programming language and its novel clock calculus that it adds to basic flow types. In this chapter, we outline how Prelude’s clock calculus may be encoded in ATS with the help of dependent types. This both simplifies the process of type-checking but also provides some possible enhancements to Prelude clocks.

Specifically, we propose adding some notion of dependent types into Prelude itself. We do not accomplish this in our modified Prelude compiler, but it does serve as an interesting path for future work. In order to distinguish from Prelude, we call our modified compiler Overture. It is our goal that this project may inspire other efforts to use ATS as a back end to support advanced features like dependent types in other simply typed languages in the future.

With that goal in mind, we can outline the encoding of Prelude clocks into the ATS type system. One very useful feature in ATS is notion of an abstract type. Abstract types are those whose direct implementation (i.e. how they are represented in a program) is unknown. For example, we could provide the following declarations to work with flows and flows with strictly periodic clocks in ATS. We declare two different types because distinguishing between the two is a requirement for checking valid use of transformation operators.

```
abstype Flow (a:type)
abstype StrictFlow (a:type)
```

We do not wish to compile these ATS programs to running code. Instead, we utilize
type-checking to verify Prelude programs’ correctness. Therefore, having abstract types at our disposal allows us to define and use types without providing any implementation details. This, along with the ability to augment any type with static annotations, gives us a strong framework for verifying programs in other languages where types may not have these “static annotations” come from the statics in ATS and provide the language’s support for dependent types. In the example above you can see one such annotation already. Since every flow is a stream of values of a single type, we attribute some a type $a$ to every flow. This makes flows polymorphic and also restrict functions to accept only flows of certain types. In the next section, we will see how we can put this and more advanced restrictions in place in our programs.

### 3.1 Dependent Types in ATS

Dependent types in ATS come from the Applied Type System (ATS) logical framework[9], which gives the language its name. They share a similarity with the more restricted form found in Dependent ML[11] which predates ATS. In the Applied Type System, each program element (type, value, or sort) is either static or dynamic. We refer to dynamics as any element belonging to the dynamic portion of the program and likewise we use the term statics.

Statics form the language of dependent types in ATS whereas dynamics are concrete values that occur in run-time programs. It is important to stress that statics have no representation in the final compiled program. They exist in the language as a means for us to enforce precision in program behavior. The way we relate the two is by indexing dynamic types with static types so that we may capture program invariants. When we construct a program, we can use predicates over these indices to enforce certain behavior. During type-checking, expressions that have predicates over static indices will generate constraints. These constraints are fed to an automated solver which searches for a counter example to the constraint in the current program context.
To see this process in action, let us consider the simple example of the list data type found in most functional programming languages.

```plaintext
datatype list (a: type) =
  | nil (a) of ()
  | cons (a) of (a, list(a))
```

For someone familiar with recursively defined data types, this is fairly straightforward. Suppose we wanted to statically reason about the length of this list. For example, we would like to implement a `length` function that we may prove yields the length of the list.

In order to accomplish this goal, we must encode the length of a list into the type definition. Using statics, we do this in such a way that adds no run-time overhead in the final program. Let us refine our previous definition of list to be more precise with respect to the length of the list. This is a demonstration of using Guarded Recursive Datatype Constructors[10].

```plaintext
datatype list (a: type, n: int) =
  | nil (a, 0) of ()
  | {n: int | n >= 0}
      cons (a, n+1) of (a, list(a, n))
```

The first constructor `nil` is straightforward as any empty list must have a length of 0. We introduce some new syntax in this example in order to create a type guard over the `cons` constructor. When we define an inductive constructor, we define a new instance of a type in terms of another instance. If we wish to capture the length of the list in this type, we need to express it in terms of the list we are adding two. This requires a new variable `n` that is universally quantified subject to the constraint that `n >= 0`. Indeed, a list of non-negative length makes little sense at all. With a universally quantified term `n` we can precisely capture the length of the list produced by `cons` as `n + 1`. This is a somewhat trivial example, but it is an easy way to understand how one may use the statics of ATS to enforce precision in a program using types.
Using a slightly more formal syntax, we can express the types of the above constructors as follows:

\[ n i l : \forall a : type \quad () \supset () \rightarrow list(a, 0) \]
\[ cons : \forall a : type, \forall n : int \ (n \geq 0) \supset (a, list(a, n)) \rightarrow list(a, n + 1) \]  

(3.1)

Where the \( \supset \) symbol denotes that the type given on the right hand side is “guarded” by the static expression on the left hand side. We use the term guarded to mean that whenever this construct is used in a program, the static expression in the guard will be checked for validity by a constraint solver.

Note, we can use some short hand to introduce a new sort in the ATS statics to represent natural numbers. This allows us to implicitly include the guard expression that \( n \) must be greater than or equal to zero in the previous examples.

\[ nat : \forall n : int \ n \geq 0 \]  

(3.2)

Now, we can demonstrate the process of type checking expressions with an example. Let us implement a length function that has the following type:

\[ length : \forall a : type, \forall n : nat \supset list(a, n) \rightarrow int(n) \]  

(3.3)

The meaning of the functions return type we want to express is an integer whose value is the length \( n \) of the list we give. The expression \( int(n) \) is an example of a singleton type as any integer with this type is restricted to the single value of \( n \). This restriction helps us enforce precision and demonstrates ATS’ flavor of dependent types. Using \textbf{length} we can produce an integer whose value is dependent on the list we gave it while introducing
no run-time overhead.

When we go to implement `length` we are obligated to produce an integer whose value is equal to length of the list. The following simple implementation achieves this goal.

```hs
implement {a} length {n} (xs) =
  case + xs of
  | nil () => 0
  | cons (x, xss) => 1 + length (xss)
```

We now present how the type checker processes `length`. A type judgment in ATS consists of a context of static variables \( \sigma \), a set of propositions \( A \), and a contract that binds some expression \( e \) to some type \( T \). The duty of the type checker is to determine whether the contract is valid assuming all the propositions in \( A \), which may contain variables in \( \sigma \) and static constants. The sorts of such constants are limited to either \( bool, int, address \), or \( class \). Constants may also be functions whose signature contains these sorts.

Each of the branches in the above example consist of a single expression, which means we need to make two type judgments to verify `length`. For the `nil` constructor, we have the following

\[
\sigma = \{a, n\}
\]

\[
A = n \geq 0, n = 0
\]

\[
0 : int(n)
\]

If we consider the constant 0 to be of type \( int(0) \), then this is equivalent to determining whether \( 0 = n \) is valid for all possible assignments of \( n \). We see here that \( n = 0 \) is an assumption, and so no such counter-example exists.

In general, we need a decision procedure that can take the context, assumptions, and contract \( e : T \) and determine whether it is valid. The contract is valid if and only if \( \neg e : T \) is unsatisfiable. In the next chapter, we will elaborate on this further where we discuss the
new constraint solver backed by an SMT solver.

Since the base case is valid, we can repeat this exercise for the inductive case with the \texttt{cons} constructor. When we do pattern matching on \textit{xs} we get the head of \textit{xs} and its tail. In addition, we now have an extra static variable at our disposal, the length of \textit{tail(xs)} and the assertion that \( n = m + 1 \) assuming \( m \) is the length of \textit{tail(xs)}. Recall that this assertion is given from our definition of the list type. This gives us the following for the type judgment.

\[
\begin{align*}
\sigma &= \{a, n, m\} \\
A &= \{n \geq 0, n = 1 + m\} \\
1 + \text{int}(m) : \text{int}(n)
\end{align*}
\]

From the assertion it is straightforward to see that this is valid. Since both branches type check to \text{int}(n), the type checker has proven that for all lists, our function yields the length of \textit{xs}.

### 3.2 Types for Prelude Operators

In the previous section, we outlined how we could use statics to restrict functions on lists to never encounter an out of bounds error for all programs that use them. The way we accomplished this was by indexing the length of a list with its type and then showed how we could use guarded types to ensure precise program behavior. In that example, we used static information of a type to precisely capture the length of the list in memory. Indeed, most uses of statics in ATS relate to a formal specification of program actions in memory. Yet, synchronous languages have no concrete representation of memory at all. Instead, we plan to use the statics in order to capture temporal properties of a Prelude program. With lists we expressed the length of a list statically with no run-time checks. By encoding Prelude programs into ATS, we will precisely encode clock information in the statics.
abstype Flow (a: type, n: int, p: rat)

abstype StrictFlow (a: type, n: int, p: rat)

As in our description of Prelude’s clocks, \( n \) and \( p \) are the period and phase offset of a flow, respectively. Borrowing the notation from the previous section, we can encode all of the clock transformation operators into ATS using the types given above.

\[
multiply: \forall a : type, \forall n, k : pos, \forall p : rat \\{ k \mid n, n \cdot p \in \mathbb{N}\} \\
\implies (\text{StrictFlow}(a, n, p), \text{int}(k)) \to \text{StrictFlow}(a, n/k, p \cdot k)
\]

\[
divide: \forall a : type, \forall n, k : pos, \forall p : rat \\{ k \mid n, n \cdot p \in \mathbb{N}\} \\
\implies (\text{StrictFlow}(a, n, p), \text{int}(k)) \to \text{StrictFlow}(a, n \cdot k, p/k)
\]

\[
shift\_phase: \forall a : type, \forall n : pos, \forall k, p : rat \\{ n \cdot (p + k) \in \mathbb{N}\} \\
\implies (\text{StrictFlow}(a, n, p), \text{rat}(k)) \to \text{StrictFlow}(a, n, p + k)
\]

\[
fby: \forall a : type, \forall n : pos, \forall p : rat \\{ n \cdot p \in \mathbb{N}\} \\
\implies (a, \text{StrictFlow}(a, n, p)) \to \text{StrictFlow}(a, n, p)
\]

We now introduce two operators \textbf{tail}, and \textbf{cons} that allow us to add to and drop values to a flow. This is similar to treating flows like linked lists, but we need to encode the effects they have on clocks.
tail : ∀a : type, ∀n : pos, ∀p : rat \{n \cdot p \in \mathbb{N}\}

\[ \implies StrictFlow(a, n, p) \rightarrow StrictFlow(a, n, p + 1) \]

cons : ∀a : type, ∀n : pos, ∀p : rat \{n \cdot p \in \mathbb{N}\}

\[ \implies (a, StrictFlow(a, n, p)) \rightarrow StrictFlow(a, n, p - 1) \]

The last operators we need to incorporate under-sample flows based on boolean flows. With the exception of \textbf{merge}, they simply produce a \textit{Flow} type as they do not have strictly periodic clocks. This disqualifies them from being transformed with any other operators as shown by the types we have presented here.

merge : ∀a : type, ∀n : pos, ∀p : rat \{n \cdot p \in \mathbb{N}\}

\[ \implies (StrictFlow(boolean, n, p), StrictFlow(a, n, p), StrictFlow(a, n, p)) \rightarrow StrictFlow(a, n, p) \]

when : ∀a : type, ∀n : pos, ∀p : rat \{n \cdot p \in \mathbb{N}\}

\[ \implies (StrictFlow(boolean, n, p), StrictFlow(a, n, p)) \rightarrow Flow(a, n, p) \]

when\_not : ∀a : type, ∀n : pos, ∀p : rat \{n \cdot p \in \mathbb{N}\}

\[ \implies (StrictFlow(boolean, n, p), StrictFlow(a, n, p)) \rightarrow Flow(a, n, p) \]
So far we have only thought of communication through flows in Prelude as single values arriving at points in time. Recall the simple example we presented earlier where a controller queries a faster task for information. When we made the communication between them synchronous, we specified that every 10th value produced by the faster task would be read by the controller.

Suppose that the controller wants all values that were produced in the last period. This requires we provide a queuing mechanism in the clock calculus. This is a feature that is outlined in Prelude’s specification, but not yet implemented in the current release of the program.

We can actually add verification support for this feature with dependent types in ATS. Suppose we have a strictly periodic flow \( f \) that we want to under-sample by some factor \( k \). Instead a flow of values each with type \( a \), we can instead think of the flow as having values of type \@\([a]\)[k], which is ATS shorthand for an array containing \( k \) elements of type \( a \). This gives us the following type for a divide operation that supports queuing.

\[
\text{divide_queue} \forall a : \text{type}, \forall n, k : \text{pos}, \forall p : \text{rat} \{k \mid n, n \cdot p \in \mathbb{N}\} \\
\subseteq (\text{StrictFlow}(a, n, p), \text{int}(k)) \rightarrow \text{StrictFlow}(\@\([a]\)[k], n/k, p \cdot k)
\]

### 3.3 ATS as a Verification Back End

With the types outlined above, we can now write ATS programs that correspond semantically to their Prelude counterparts in terms of the synchronous checks that must be performed. The process of generating ATS code from Prelude programs is fairly straightforward. All imported nodes are declared as external functions with all clocks referenced in the node being equal. This is required since imported nodes are simply external functions that have no real time knowledge of the flows they consume and produce. Nodes that are implemented are transformed into full ATS functions with the corresponding clocks.
There are two issues we need to address during this compilation. The first is a minor problem of inferring the clocks of local flows. Addressing this issue is a subject for future work as currently we require explicit clocks for local variables to be provided by the programmer. This specification is checked for correctness as with any other clock using the types we outlined in the previous section.

The second issue is that Prelude is a concurrent declarative language whereas ATS is not. This becomes evident if we try to translate a program where a flow is used as an argument and then defined later in the code. We cannot simply reorder expressions, because circular definitions are common, for instance in the simpler controller example we outlined earlier. When we translate to ATS, we make use of linear types and its theorem proving facilities[2] to make sure the flow is eventually defined and that its clock matches our expectation. As an example, recall our simple controller task that queried data from a fast running task. We had two local flows that stood for the command sent from the controller and the data sent back from the receiver. We can declare both of these flows in ATS using the \texttt{var} keyword.

\begin{verbatim}
var command : strict_flow (int, 30, Rational(0))
var response : strict_flow (int, 10, Rational(0))
\end{verbatim}

With the local flows declared this way, we can pass them as arguments to imported nodes and clock transformation functions. Eventually, they must be defined in the node. To enforce this requirement, we can use what we call linear props or views in ATS. Props are program entities that are used solely for theorem proving in ATS. As such, they introduce no run-time overhead and are completely separated from program terms in the syntax. A linear type is one that must eventually be consumed in the scope in which it is used. A view is then a prop that must be consumed. Put more simply, it is not allowed to “leak”. In this case, we want to prove that all local flows are eventually defined. To do that, we define the following abstract view.

\begin{verbatim}
absview FlowFuture
\end{verbatim}
We obtain a future with the following axiom. Its type signature means that for any flow that is uninitialized, you may consider it initialized, but you are obligated to show some evidence it is later defined. The obligation is given by the view. If the view is not consumed in the current node, then an ATS type error will occur.

\[
\text{praxi}
\]  
\[
\text{flow\_future\_make \{a: type\} \{n: nat\} \{p: rat\} (}
\]
\[
\&\text{strict\_flow (a, n, p)?} >> \text{strict\_flow (a, n, p)}
\]
\[
\): FlowFuture
\]

The syntax \&a? \&a asserts the following. Given an uninitialized reference to a variable of type \(a\), the variable becomes initialized as a consequence of this function. Now, we need a method to consume the view produced by our contract function above. When compiling a node to ATS, whenever we define a flow that is declared local, we generate a call to the following axiom with the local variable and the flow matching its definition.

\[
\text{praxi}
\]  
\[
\text{flow\_future\_elim \{a: type\} \{n: nat\} \{p: rat\} (}
\]
\[
\text{FlowFuture, \&strict\_flow (a, n, p), strict\_flow (a, n, p)}
\]
\[
\): void
\]

If the clock of the defined flow does not match the clock given to our local variable, a synchronization constraint is violated and we report an error. On the other hand, if the flow is never defined, we also catch this error because all \textit{FlowFutures} would not be properly consumed. The type of the above axiom consumes the view given to it. This completes our approach to the concurrency issue described earlier.

In our modified Prelude compiler, we successfully generate ATS code from Prelude abstract syntax trees. This process is fairly straightforward given Prelude’s sparse syntax. The system we have now is only a proof of concept. For instance, its error messages are in terms of the produced ATS program, not in terms of the original code. Nonetheless, we are able to automatically catch synchronization errors from Prelude programs using
nothing more than the ATS type checker. This demonstrates two things. First, dependent

types can enforce real time invariants. Second, ATS can be used to support an advanced
reasoning feature in a comparatively simpler typed language.

3.4 Possible Extension

There is an interesting extension we could feasibly provide to make Prelude’s clock calculus
even more powerful. We could possibly allow the expression of a flow’s clock in terms
of other clocks. This feature is missing in the current description of Prelude, but it is
foreseeable we could add this feature into the Prelude type checker.

Note, we outline here of the possibility of supporting something like dependent clocks
in the Prelude type system. We do not say here how the compiler will be able to take
such a program after type checking and perform schedulability analysis on it. This feature
should only restrict errant implementations from passing type checking and not affect the
compilation stage. If we find this idea useful, we could explore it further in future work.

To see how this would work, let us consider the following node signature in Prelude.

\[ \text{node } \{c: \text{clock}\} (i: \text{rate } c) \text{ returns } (o: \text{rate } c/2) \]

Here we borrow a bit from ATS’ syntax to express precisely \( o \)'s clock in terms of the
input clock \( c \). If we wanted to translate this restriction into an ATS type, the following
should suffice.

\[
\text{node } \forall a: \text{type}, \forall n: \text{pos}, \forall p: \text{rat } \{2 | n, n \cdot p \in \mathbb{N}\}
\]

\[
\supset \text{StrictFlow}(a, n, p) \rightarrow \text{StrictFlow}(a, n/2, p \cdot 2)
\]
Chapter 4

Enriching Dependent Types in ATS

We are mostly interested in expanding the domain of ATS’ statics to include linear rational arithmetic. Simply modifying the current ATS constraint solver could be a significant amount of effort. It would also leave us with a system only marginally better than its predecessor. At the same time, we open up the possibility of introducing bugs into the solver by supporting more kinds of constraints. For these reasons, we turn to external solvers to help improve our system.

The study of how to automatically solve mathematical formulae has yielded many practical tools in recent years. SAT solvers’ drastic jump in terms of efficiency over the past decade accompanied a new class of tools called SMT solvers. These solvers decide formulas outside the realm of boolean satisfiability and in domains of interest to software verification and constraint solving. An SMT solver pairs an efficient SAT solver with multiple first order theories. This gives these tools their name, Satisfiability Modulo Theories.

The interaction with an SMT solver is done to determine the satisfiability of some formulas. In addition, many solvers provide the ability to return a model that satisfied context of assertions. Unlike more powerful interactive theorem provers, SMT solvers are completely automated and support a wide range of theories. These range from integer and real arithmetic to fix width bit vectors, arrays. Some theories have recently been supported to even handle a notoriously complicated data type such as IEEE floats [8].

Since SMT solvers decide both linear integer and real arithmetic they are prime candidates to serve as a constraint solver for ATS. Their active use and continued development provide us with a more robust solution than writing our own. The continued development
of theories will also let us explore using more advanced sorts such as fixed width bit vectors and arrays in the ATS statics.

For the constraint solver made for this thesis, we chose Microsoft’s Z3[3] because of its excellent performance in SMT-COMP. In an earlier prototype that we built, we found that using Yices 2[4] gave performance that frequently beat the custom constraint solver currently used in ATS. A path for future work is to try out the recently released Yices 2 as an external constraint solver and compare it to our current one that utilizes Z3.

4.1 Requirements for a Constraint Solver

In order for a decision procedure to solve constraints generated by ATS programs, it must implement several key functions. Part of this work was done to identify exactly what could be optionally be removed from the ATS compiler and deferred to an external program. This opens up the possibility for prototyping many different implementations that need not be tied to any single language or interface. By exporting program constraints in plain text via JSON, we provide a good deal of flexibility on the tools that may take on the role of constraint solver.

In this section, we outline the interface any external will need to support in order to act as a constraint solver for ATS. Our goal in providing this interface is that it may inspire an effort to use other formal reasoning tools including interactive theorem provers like Coq, Isabelle, or ACL2. With the ability to add new sorts to the statics, we could imagine constraint solvers built for specific classes of programs. So far, the statics in ATS have been a very general purpose tool, but in future work we would like to explore what kinds of program properties we may verify semi-automatically using tools even more powerful than SMT solvers behind the ATS statics.

The first requirement of a solver is the ability to keep a context of static variables. The essential sorts variables may have are simply \textit{int} and \textit{bool}. Next, propositions need to be created and asserted in this context. The context must support scopes. That is, we
must be able to add new scopes to the context and discard old ones like a stack. When pop is encountered, all variables and asserted propositions are discarded from the current context. Finally, we must be able to query whether or nor a context is valid. That is, there exists no model or assignment to variables in the context causing a proposition to be false. SMT solvers naturally support all of these requirements. As an example, we include the following interaction with an SMT solver where we query whether the formula $x + y > 0$ is valid.

\begin{verbatim}
(declare-fun x () Int)
(declare-fun y () Int)

(push)
(assert (> x 0))
(assert (> y 0))

(push)
(assert (not (> (+ x y 0))))
(check-sat)

(pop)
(pop)
\end{verbatim}

4.2 Adding New Sorts to ATS Statics

ATS provides an interface to add new sorts to the statics. This, coupled with support for external constraint solving allows one to expand the verification domain of ATS without actually modifying the compiler itself. This truly make ATS, in our opinion, a potential platform for program verification.
In this section, we demonstrate the interface available by adding support for rational numbers in ATS statics. These provide the missing functionality presented in our ATS representation of Prelude clock operators. Recall that earlier in our discussion of dependent types we mentioned that static terms are limited to the few sorts bool, int, and address. Indeed, the default constraint solver is limited to linear integer arithmetic, so any constraints generated from type indices of any other sort would simply be rejected.

A recent addition to the ATS compiler allows us to export all the constraints from an ATS program into a JSON format. ATS does provide primitives for declaring abstract sorts in the statics. With an external solver, this allows us to expand the vocabulary of the statics to what an SMT solver can understand. To see the potential of this, let us see what is required to add support for real numbers in the statics.

First, we declare an algebraic sort rat.

```
datasort rat =
   | Rational of (int)
   | RationalDiv of (int, int)
```

This provides us two ways to express constants as rational numbers in programs. The first constructor just casts any integer to a rational number. The second creates an expression \( p/q \) for \( \text{RationalDiv}(p, q) \). From the perspective of the constraint solver, these two constructors are just static constants with a function sort. That is, when we create a rational number in the statics, the constraint solver sees an application of a static constant to the arguments. For example, if we have \( \text{RationalDiv}(2, 3) \) in a program, the corresponding expression sent to the constraint solver would be something like the following

```
apply(RationalDiv, (2, 3))
```

The constraint solver then needs to transform this into something it (the SMT solver) understands. Within the constraint solver, we define a map from static constants such as \( \text{RationalDiv} \) to functions that compile their application into an SMT formula. For
any static constant that lacks a definition in this map, we simply create an uninterpreted function. For all the functions over rationals we need, we simply use the SMT solver’s API to turn each static function call into an equivalent SMT formula. As an example, consider the following interface for static rationals. We use stacst to declare the sort of each function and then stadef provides us convenient shorthand and overloads standard operators.

```
stacst mul_rat_rat : (rat, rat) -> rat
stadef * = mul_rat_rat

stadef / = div_rat_rat

stadef + = add_rat_rat

stadef - = sub_rat_rat

stadef >= = gte_rat_rat

stadef is_rat_int : rat -> bool

is_rat_int (n:rat) =
    is_rat_int\(n\) && (n >= Rational(0))
```

This is truly, in our opinion, a testament to ATS’ flexibility. Without modifying the compiler, we are able to augment the statics of ATS with just the above declarations which
may be provided in a simple sats file. Files with the .sats extension in ATS are akin to header files in other languages. Once the sats file is loaded, we can now use rational numbers in the statics alongside the standard bool and int. The amount of work required to support the above functions is almost trivial in the constraint solver, as Z3’s API has straightforward methods of building arithmetic expressions. We have already experimented with adding support for fixed width bit vectors and arrays in the statics to some success.

We hope that this work and future efforts will inspire others to see ATS as a method to put verification tools directly into practice. SMT solvers are very powerful tools, but using them in program verification is complicated because you often must model your program in the language of the solver. This is likely true not only with SMT solvers, but with any verification framework. ATS, on the other hand, allows you to apply parts of your model directly to your program. In this thesis, our model included clocks and the constraints that must be met for them to be used properly. ATS allowed us to encode their properties into the type system and relate the types we use for verification to program types. As we used these types in the program, ATS extracted all the constraints on our behalf, which we then could feed almost directly to an SMT solver. ATS allows you to embed the formal specification of a program directly into its implementation. There are many tools used today for verifying formal properties of systems. With ATS and the methods outlined here, we hope that we can put them to work.
Chapter 5

Related Work

The Liquid Types[7] project is in many ways similar to our case study of using ATS as a verification back end. In fact, Liquid Types have been used in a wide array of languages from OCaml, C, to Haskell where they provide safety guarantees without the expense of cumbersome manual annotations to source code. In this thesis, we explore using the advanced features of ATS as a foundation for similar features in a more simply typed programming language.

At first glance, both of these projects seek to solve the same problem. That is, there is an absence of formal verification in many programming languages. Indeed, both are rooted in dependent types as put forth by Dependent ML. Liquid types primarily focuses on the issue of inference of formal properties to make type annotations unneeded. In contrast, ATS provides very little in the way of inference.

This more restricted form adds a good deal flexibility to ATS in that it poses little restriction on what kinds of properties may be checked. That is, it is up to the user to augment their dynamic code with statics they deem fit enough to enforce precision in their program. In this thesis, we used this framework to expand the decision power of ATS statics to new domains while using them to verify real-time properties of a system.

The Boogie Intermediate Verification Language bears some strong resemblance to how we use ATS in this project. Boogie forms the foundation of verification engines used for languages such as C, C#, and Microsoft’s own Dafny language. In these systems, a program’s specification is extracted from program source and transformed invariants understood by Boogie. Boogie then checks these invariants using an SMT solver such as
Z3. Our use of ATS in this thesis shares a similar approach. It is the subject of future work to perform a more in depth comparison between ATS and Boogie as verification back ends.
Chapter 6

Conclusions

By compiling Prelude to ATS we show that ATS can be used as a verification back end to provide advanced features to a simply typed language. Another simply typed language lacking in any real verification features is C. With the external constraint solver we developed for this project, we would like to explore offering an extension to C that allows a programmer to make use of dependent types, linear types, and views which are a feature present in the theorem proving component of ATS. Overall, we feel that providing facilities for theorem proving in a C compiler could help system programmers build more reliable systems. Since the semantics of C are so close to ATS, we feel this could demonstrate the usefulness of program verification in a more familiar environment for engineers.
Bibliography


