Typed Abstract State Machines

Giuseppe Del Castillo
(Universität-GH Paderborn, Germany
giusp@uni-paderborn.de)

Yuri Gurevich
(University of Michigan, USA
gurevich@eecs.umich.edu)

Karl Stroetmann
(Siemens AG, Germany
Karl.Stroetmann@mchp.siemens.de)

Abstract: As introduced in the Lipari guide, Abstract State Machines (abbreviated as ASMs) are untyped. This is useful for many purposes. However, typed languages have their own advantages. Types structure the data, type checking uncovers errors. Here we propose a typed version of ASMs.
Key Words: abstract state machine, type system, polymorphism
Category: D.3.1, F.1.1, F.3.2, F.3.3

1 Introduction

In programming languages, there is a clear tendency towards typed languages. `BCPL`, the untyped forerunner of `C`, has been completely replaced by `C` and `C++`. Similarly `Lisp` is being gradually replaced by `ML` and `Haskell`. Concerning logic programming, the last years have seen the development of several typed logic programming languages, e.g., `Mercury` [HSC96], `Protos-L` [Bei95a], and `Gödel` [HL92]. Among others, there are the following important reasons for the introduction of types:

1. To facilitate structuring of the data of an application.
2. To have a type checker that automatically detects errors at compile time.

Detecting errors automatically when writing a program is certainly useful. However, the real value of type checking often becomes apparent in the maintenance phase of the software life cycle. It is here that the type checker reveals dependencies between different parts of a program that may easily be overlooked otherwise.

There are similar reasons for adding types to specification languages. Furthermore, following the *object oriented* software engineering approach one starts building a complex system by building the *object model*. It is obvious that this process is facilitated by a suitable type system. However, typing a specification language is not without its own pitfalls; cf. [LP97]. The same richness of types that makes it convenient to describe types in applications may become an impediment. In the worst case, it may lead to inconsistencies of the type system. [LP97] cites some examples. Also, for a rich type system the well-typedness problem tends to get hard and might even be undecidable. And then one of the main typing benefits is lost.

On the other hand, in a weak type system, some functions that would be total if only their domains could be expressed as types become partial. With most specification languages, the notion of a partial function is difficult to handle. In fact, much of the intricacies of modern specification languages have their origin in the treatment of partial functions. Fortunately, in the framework of ASMs [Gur95] this problem has been solved: formally speaking, all functions are total. This is achieved by providing a default value. As a consequence, the richness of a type system is less vital for ASMs than it is for many other specification languages.

For these reasons, we propose a simple type system for ASMs. We present a type system that introduces parametric polymorphism as suggested by [Mor68] and [Mil78]. Our type system can be described as the type system of the programming language ML [MTH90] restricted to the first order case and without the *let*-construct. A followup paper [GS98] extends our type system in several ways including *casting* [FM90] and a modest version of type classes [NHN80, WB89].

This paper is organized as follows. Section 2 introduces types and terms. The notion of a typeable term is given and we discuss the concept of a principal type. Section 3 presents typed ASMs. Section 4 elaborates the previously introduced concepts with an example. Finally, Section 5 is a concluding discussions.

This is not the first paper on typed ASMs. In the pioneering paper [Zam97], Zamulin presented a different and more ambitious approach. We discuss his work in Section 5.

In order for this paper to be self contained, we give, in the appendix, an algorithm for computing the principal type of a typeable term. (This algorithm is based on unification [Rob65].) Since this paper is concerned mainly with the concept of a typed ASM, we do not discuss the implementation of the algorithm.) There is no pretence on originality here; the algorithm can be derived from the relevant text books, e.g., [Hin96, Mit96]. However, the derivation may be not completely obvious to some readers interested in ASMs but not familiar with lambda calculus. Our unpretentious appendix may be useful to them.

There was a division of labour among the authors of this paper. The second and third authors did the theoretical part. The first author implemented the type checking algorithm [Cas98] and provided the example in Section 4.

2 Types and Terms

In this section we define the syntax of terms and types, define typeable terms, and present the amount of type theory that is needed for the development of typed ASMs. For the readers convenience we prove a number of known facts, for example the uniqueness of principal types is proved. Of course, this is standard material that can also be found in text books dealing with type theory, e.g., [Hin96, Mit96].

2.1 Syntax of Types

Definition 1 (Type Vocabulary, Types) A *type vocabulary* is a pair $\langle \mathbb{T}, \mathbf{arity} \rangle$ such that $\mathbb{T}$ is a set of *type constructors* and **arity** is a function

$$\mathbf{arity}\colon \mathbb{T} \to \mathbb{N}$$

assigning an *arity* to every type constructor $F \in \mathbb{T}$.

Fix an infinite list of *type parameters*. Then the *types* of the given type vocabulary are defined inductively:

1. Every type parameter is a *type*.
2. If F is an n-ary type constructor and $\sigma_1, \ldots, \sigma_n$ are *types*, then $F(\sigma_1, \ldots, \sigma_n)$ is a *type*.

The set of *type parameters* used in a type σ is denoted by $\mathrm{Par}(\sigma)$. If $\mathrm{Par}(\sigma) = \emptyset$, then σ is called a *closed* type, otherwise σ is called a *generic* type. $\diamond$

Notation: We denote type parameters by capital letters S and T. Type constructors are denoted by words written in lower case and set in typewriter style, i.e., `int`, `float`, or `char` are examples of type constructors. The lower case Greek letters π, ϱ, σ, and τ are used to denote types. $\diamond$

Example 1 Consider a type vocabulary

$$\langle \{\texttt{int}, \texttt{float}, \texttt{bool}, \texttt{string}, \texttt{list}, \texttt{dictionary}\}, \mathbf{arity} \rangle,$$

where **arity** is defined as follows:

$$\begin{array}{l} \mathbf{arity}(\texttt{int}) := 0, \quad \mathbf{arity}(\texttt{float}) := 0, \quad \mathbf{arity}(\texttt{bool}) := 0, \\ \mathbf{arity}(\texttt{string}) := 0, \quad \mathbf{arity}(\texttt{list}) := 1, \quad \text{and} \\ \mathbf{arity}(\texttt{dictionary}) := 2. \end{array}$$

Then the following constructions are types:

$$\begin{array}{l} \texttt{int}, \quad \texttt{list(int)}, \quad \texttt{dictionary}(\texttt{string}, \texttt{list}(\texttt{int})), \\ T, \quad \texttt{list}(T), \quad \text{and} \quad \texttt{dictionary}(\texttt{string}, T) \end{array}$$

The constructions appearing in the first line are all closed types, while for any type σ in the second line we have $\mathrm{Par}(\sigma) = \{T\}$. $\diamond$

2.2 Vocabulary

Definition 2 (Vocabulary) A *vocabulary* is a tuple

$$\mathcal{V} = \langle \mathbb{T}, \mathbf{arity}, \mathbb{F}, \texttt{Profile} \rangle$$

where $\langle \mathbb{T}, \mathbf{arity} \rangle$ is a type vocabulary and the components $\mathbb{F}$ and `Profile` satisfy the following:

1. $\mathbb{F}$ is a set the elements of which are called *function symbols*.
2. `Profile` is a function assigning a *profile* to every function symbol $f \in \mathbb{F}$:

$$f : \sigma_1 \times \ldots \times \sigma_n \to \tau$$

Here $\sigma_1, \ldots \sigma_n, \tau$ are types. If $n = 0$ we write $f : \tau$ instead of $f :\to \tau$.

The function Par computing the type parameters of a type is extended to profiles. We have

$$\mathrm{Par}(\sigma_1 \times \ldots \times \sigma_n \to \tau) := \mathrm{Par}(\sigma_1) \cup \ldots \cup \mathrm{Par}(\sigma_n) \cup \mathrm{Par}(\tau).$$

We call a function symbol f *polymorphic* iff $\mathrm{Par}(\texttt{Profile}(f)) \neq \emptyset$. If the profile of f is $\sigma_1 \times \ldots \times \sigma_n \to \tau$, then n is called the *arity* of f.

We assume that an infinite list of *variables* is fixed; variables are distinct from function symbols and type parameters. $\diamond$

The definition of a vocabulary given above does not assign types to variables. Typing the variables would have been a reasonable choice. In that case, instead of a single list of variables, we would have a family of lists of variables indexed by types. Practically speaking, we would be forced to declare the type of every variable.

Notation: If $\sigma_1 \times \ldots \times \sigma_n \to \tau$ is a profile, then this profile is also written as $\boldsymbol{\sigma} \to \tau$, i.e., we use the boldface $\boldsymbol{\sigma}$ to denote $\sigma_1 \times \ldots \times \sigma_n$. $\diamond$

We require that every vocabulary contains a nullary type constructor `bool` and the following *logic function symbols*: "**true**", "**false**", "$\neg$", "$\wedge$", "$\vee$", "$\to$", "**default**", and "=". The profiles of these symbols are given as follows:

$$
\begin{array}{ll}
\textbf{true} & : \texttt{bool}, \\
\textbf{false} & : \texttt{bool}, \\
\neg & : \texttt{bool} \to \texttt{bool}, \\
\wedge & : \texttt{bool} \times \texttt{bool} \to \texttt{bool}, \\
\vee & : \texttt{bool} \times \texttt{bool} \to \texttt{bool}, \\
\to & : \texttt{bool} \times \texttt{bool} \to \texttt{bool}, \\
= & : T \times T \to \texttt{bool}, \\
\textbf{default} & : T.
\end{array}
$$

Above, T is a type parameter, therefore the function symbols "=" and "**default**" are polymorphic. The intention is that T ranges over all closed types. Therefore the profile for "=" is interpreted as a short hand notation for the following set of profiles

$$\{= : \ \tau \times \tau \to \texttt{bool} \mid \tau \text{ is a closed type}\}.$$

This interpretation shows that the identity of the type parameter T is irrelevant: The semantics of the profile $=: T \times T \to \texttt{bool}$ is the same as the semantics of $=: S \times S \to \texttt{bool}$ for a type parameter S different form T.[1]

In the rest of this paper we assume that a vocabulary satisfying the requirements given above has been fixed.

Note: Before proceeding, we should discuss the function symbol **default** and its profile. The intention is that for every closed type the function **default** denotes a corresponding default value. We stipulate that, in the case of the type `bool`, **default** denotes the value **false**. The user is free to choose other default values. In the case of the type `int` (of integers) one may choose to interpret **default** as 0. Alternatively, it may be more convenient to extend the set of integers with a new element called **default**. (The latter is especially convenient if one deals with partial functions.) Suppose we have chosen the integer **default** to be 0. Then both **default** = **false** and **default** = 0 are true. However, we may not be mislead into concluding **false** = 0. The reason is that the function **default** has a meaning only if its type has been fixed. As **default** has the polymorphic

[1] In order to stress that the identity of the type parameter T appearing in the profile of "=" does not matter, we could have used the notation

$$= : \ \forall T(T \times T \to \texttt{bool}).$$

For example, this is done in the literature on the programming language `ML` [MTH90]. For notational simplicity, we make these universal quantifiers implicit.

profile T, we have to determine the context in which **default** appears in order to instantiate the type parameter T. For example, in the equation **default** = **false** the function profile of "=" forces us to interpret T as `bool`. If the profile of 0 is `int`, then T has to be interpreted as `int` in the equation **default** $= 0$.

The *overloading* of **default** is similar to the overloading of the empty list. Suppose we have polymorphic function symbols `nil` and `cons` in our vocabulary:

$$\begin{aligned} &\texttt{nil} &&: \texttt{list}(T) \\ &\texttt{cons} &&: T \times \texttt{list}(T) \to \texttt{list}(T) \end{aligned}$$

Here the symbol `nil` denotes the empty list, while $\texttt{cons}(x, l)$ inserts the element x at the beginning of the list l. Then the meaning of `nil` is only determined by its context: `nil` could denote both the empty list of integers and the empty list of Booleans. It is reasonable to assume that the empty list of Booleans is different from the empty list of integers. ◇

2.3 Syntax of Terms

The notion of a term is defined inductively: variables are terms and if f is an n-ary function symbol and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term, too. If t is a term, then the set of variables occurring in t (denoted $\mathrm{FV}(t)$) is defined by induction on t as usual. A term t is a *closed* term iff $\mathrm{FV}(t) = \emptyset$.

Notation: We denote variables by the lower case letters x, y, z. We also use lowercase strings like xs and ys to denote variables. f is used as a meta-variable to range over function symbols. Concrete function symbols are denoted by strings written in lower case and set in type write style, i.e., `append` is an example of a function symbol. Terms are denoted by the letters s and t. ◇

Example 2 If `append` is a function with profile

$$\texttt{append}\colon \texttt{list}(T) \times \texttt{list}(T) \to \texttt{list}(T),$$

then the following strings are terms:

$$\texttt{append}(\texttt{nil}, xs), \quad \texttt{append}(\texttt{cons}(x, xs), ys). \qquad \diamond$$

In order to define the notion of a *typeable* term, we introduce *parameter substitutions*. A *parameter substitution* Θ is a finite set of pairs of the form

$$\left[T_1 \mapsto \tau_1, \ldots, T_n \mapsto \tau_n\right]$$

where $T_1, \ldots, T_n$ are distinct type parameters and $\tau_1, .., \tau_n$ are types. We call the set $\{T_1, \ldots, T_n\}$ the *domain* of Θ (denoted $\mathbf{dom}(\Theta)$).

A parameter substitution $\Theta = \left[T_1 \mapsto \tau_1, \ldots, T_n \mapsto \tau_n\right]$ is interpreted as a function mapping type parameters to types as follows:

$$\Theta(T) \;:=\; \begin{cases} \tau_i & \text{if } T = T_i; \\ T & \text{otherwise.} \end{cases}$$

This function is extended to types homomorphically:

$$\Theta\big(F(\sigma_1, \ldots, \sigma_n)\big) \;:=\; F\big(\Theta(\sigma_1), \ldots, \Theta(\sigma_n)\big).$$

We use a postfix notation to denote the result of evaluating Θ on a type τ,

i.e., we write $\tau\Theta$ instead of $\Theta(\tau)$. The application of a parameter substitution Θ to a profile is defined as expected:

$$(\sigma_1 \times \ldots \times \sigma_n \to \tau)\Theta = \sigma_1\Theta \times \ldots \times \sigma_n\Theta \to \tau\Theta.$$

If σ and τ are types such that $\sigma\Theta = \tau$ holds for some parameter substitution Θ, then τ is called an *instance* of σ. In this case we also say that σ is *more general than* τ (denoted $\sigma \succeq \tau$). Instances of profiles are defined similarly.

Example 3 We have $\texttt{list}(T) \succeq \texttt{list}(\texttt{int})$ since

$$\texttt{list}(T)[T \mapsto \texttt{int}] = \texttt{list}(\texttt{int})$$ ◇

If Θ_1 and Θ_2 are parameter substitutions, then their *composition* $\Theta_1 \circ \Theta_2$ is defined such that $T(\Theta_1 \circ \Theta_2) = (T\Theta_1)\Theta_2$ holds for all type parameters T. If Φ_1, Φ_2, and Θ are parameter substitutions such that $\Phi_2 = \Phi_1 \circ \Theta$, then Φ_1 is *more general than* Φ_2 (denoted $\Phi_1 \succeq \Phi_2$).

Definition 3 (Appropriate) A profile $\boldsymbol{\sigma} \to \tau$ is *appropriate* for a function symbol $f \in \mathbb{F}$ iff it is an *instance* of $\texttt{Profile}(f)$, i.e., if there exists a parameter substitution Θ such that

$$(\boldsymbol{\sigma} \to \tau) = \texttt{Profile}(f)\Theta.$$

A profile $\boldsymbol{\sigma} \to \tau$ is *closed* if it contains no parameters. ◇

Example 4 Since $\texttt{Profile}(=)$ is equal to $T \times T \to \texttt{bool}$, we conclude that both the profiles

$$\texttt{int} \times \texttt{int} \to \texttt{bool} \quad \text{and} \quad \texttt{float} \times \texttt{float} \to \texttt{bool}$$

are appropriate for "=". ◇

Definition 4 (Type Assignment) A *type assignment*

$$\Gamma = \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$$

maps distinct variables $x_1, \ldots, x_n$ to types $\tau_1, \ldots, \tau_n$ respectively. ◇

If Γ is a type assignment, x is a variable and τ is a type, then the type assignment $\Gamma, x : \tau$ is defined as follows:

$$(\Gamma, x : \tau)(y) \;:=\; \begin{cases} \tau & \text{if } y = x, \\ \Gamma(y) & \text{otherwise.} \end{cases}$$

If Γ_1 and Γ_2 are type assignments that agree on the intersection of their domains, then the *union* of these type assignments is defined as follows:

$$(\Gamma_1 \cup \Gamma_2)(x) \;:=\; \begin{cases} \Gamma_1(x) & \text{if } x \in \mathbf{dom}(\Gamma_1), \\ \Gamma_2(x) & \text{if } x \in \mathbf{dom}(\Gamma_2). \end{cases}$$

If Θ is a parameter substitution and Γ is a type assignment, then the type assignment $\Gamma\Theta$ is defined as $(\Gamma\Theta)(x) \;:=\; \Gamma(x)\Theta$. Furthermore, we define

$$\mathrm{Par}(\Gamma) \;:=\; \bigcup\{\mathrm{Par}(\Gamma(x)) \;:\; x \in \mathbf{dom}(\Gamma)\}.$$

A *type annotation* is a pair $t : \tau$ where t is a term and τ is a type.

We proceed to define the notion of a *typeable term*.

Definition 5 (Typeable Term) First we define when a type assignment Γ *entails* a type annotation $t : \tau$, symbolically $\Gamma \vdash t : \tau$.

1. If $\Gamma(x) = \tau$, then
$$\Gamma \vdash x : \tau.$$
2. If $\Gamma \vdash s_i : \sigma_i$ for all $i = 1, \ldots, n$ and the profile $\sigma_1 \times \ldots \times \sigma_n \to \tau$ is appropriate for f, then
$$\Gamma \vdash f(s_1, \ldots, s_n) : \tau.$$

Now, a term t is *typeable* iff there exist a type assignment Γ and a type τ such that $\Gamma \vdash t : \tau$. ◇

Example 5 The terms $\mathtt{append}(\mathtt{nil}, xs)$ and $\mathtt{append}(\mathtt{cons}(x, xs), ys)$ are both typeable: It suffices to define the type assignment Γ as follows:
$$\Gamma = \{x : T, xs : \mathtt{list}(T), ys : \mathtt{list}(T)\}.$$ ◇

Note: The typeability or untypeability of a term may be not completely obvious. For example, (i) $\mathtt{cons}(x, x)$ is untypeable but (ii) $\mathtt{cons}(\mathtt{nil}, \mathtt{nil})$ is typeable. To prove (i), assume that $\Gamma \vdash \mathtt{cons}(x, x) : \tau$ and let $\sigma = \Gamma(x)$. Recall that the profile of $\mathtt{cons}$ is $T \times \mathtt{list}(T) \to \mathtt{list}(T)$. It follows that the profile $\sigma \times \sigma \to \tau$ is appropriate for $\mathtt{cons}$ and thus there is a parameter substitution Θ such that $T\Theta = \sigma = \mathtt{list}(T)\Theta$. But this is impossible because the depth[2] of the type $\mathtt{list}(T)\Theta$ exceeds that of $T\Theta$. To prove (ii), recall that the profile of $\mathtt{nil}$ is $\mathtt{list}(T)$. Let Γ be the empty type assignment, $\sigma_1 = \mathtt{list}(\mathtt{int})$ and $\sigma_2 = \tau = \mathtt{list}(\sigma_1)$. Clearly σ_1 and σ_2 are appropriate for $\mathtt{nil}$ and therefore Γ entails $\mathtt{nil} : \sigma_1$ as well as $\mathtt{nil} : \sigma_2$. But $\sigma_1 \times \sigma_2 \to \tau$ is appropriate for $\mathtt{cons}$. Hence $\Gamma \vdash \mathtt{cons}(\mathtt{nil}, \mathtt{nil}) : \tau$.

Notice that the two occurrences of $\mathtt{nil}$ were treated differently from the two occurrences of x. A type assignment cannot assign different types to different occurrences of the same variable. On the other hand, different occurrences of the same function symbol may come with different appropriate profiles. ◇

The following Lemma is an immediate consequence of Definition 5 and of Definition 3 of a profile being appropriate for a function symbol.

Lemma 6 Suppose that $t = f(s_1, \ldots, s_n)$ and $f : \sigma_1 \times \ldots \times \sigma_n \to \tau$. Then $\Gamma \vdash t : \pi$ iff there exists a parameter substitution Θ such that $\tau\Theta = \pi$ and $\Gamma \vdash s_i : \sigma_i\Theta$ for all $i = 1, \ldots, n$. □

As it stands, the type of a typeable term is not unique. There is an obvious example for a typeable term that has many different types. Consider the function symbol $\mathtt{nil}$ representing the empty list. Its profile is given as

$\mathtt{nil}$: $\mathtt{list}(T)$.

According to Definition 5, the term $\mathtt{nil}$ has, among others, the types $\mathtt{list}(\mathtt{int})$, $\mathtt{list}(\mathtt{bool})$, and $\mathtt{list}(T)$. Obviously, the type $\mathtt{list}(T)$ is more general than the other types, we have
$$\mathtt{list}(T) \succeq \mathtt{list}(\mathtt{bool}) \quad \text{and} \quad \mathtt{list}(T) \succeq \mathtt{list}(\mathtt{int}).$$

[2] The *depth* of a type can be defined formally by a straightforward inductive definition.

For pairs consisting of a type assignment and a type we define the relation "$\succeq$" as follows: We have $\langle \Gamma, \sigma \rangle \succeq \langle \Delta, \tau \rangle$ iff there exists a parameter substitution Θ such that $\Gamma\Theta = \Delta$ and $\sigma\Theta = \tau$.

Now we can give the definition of a *principal type*.

Definition 7 (Principal Type) A type σ is a *principal type* of a term t if there exists a type assignment Γ such that:

1. $\mathbf{dom}(\Gamma) = \mathrm{FV}(t)$.
2. $\Gamma \vdash t : \sigma$.
3. If $\Delta \vdash t : \tau$ and $\mathbf{dom}(\Delta) = \mathrm{FV}(t)$ holds, then $\langle \Gamma, \sigma \rangle \succeq \langle \Delta, \tau \rangle$.

An appropriate Γ is a *principal type assignment* associated with the annotation $t : \sigma$. ◇

Example 6 The type $\mathtt{list}(T)$ is a principal type of the term

$$\mathtt{append}(\mathtt{cons}(x, xs), ys).$$

An appropriate principal type assignment is given as

$$\{x : T, xs : \mathtt{list}(T), ys : \mathtt{list}(T)\}.$$ ◇

Note: If σ is a principal type of a term t and Γ is a principal type assignment associated with the annotation $t : \sigma$, then Γ is not unique. To see this, consider the term $x = x$. We have

$$\{x : T\} \vdash (x = x) : \mathtt{bool} \quad \text{and} \quad \{x : S\} \vdash (x = x) : \mathtt{bool},$$

showing that both $\Gamma = \{x : T\}$ and $\Delta = \{x : S\}$ are principal type assignments associated with the annotation $(x = x) : \mathtt{bool}$.

The principal type of a term is not unique either. For example, both $\mathtt{list}(T)$ and $\mathtt{list}(S)$ are principal types for $\mathtt{nil}$. ◇

We show that a principal type and an associated principal type assignment are unique up to renaming of parameters. In order to prove this, we need the following lemma.

Lemma 8 (Uniqueness) *If $\sigma \succeq \tau$ and $\tau \succeq \sigma$, then there exists a renaming Θ of type parameters such that $\sigma\Theta = \tau$.*

Proof: Since $\sigma \succeq \tau$, there exists a substitution Θ with $\sigma\Theta = \tau$. Moreover Θ can be chosen so that its domain contains only those parameters that occur in σ. We claim that Θ is a renaming of parameters, in other words,

1. for every parameter T, we have that $T\Theta$ is a parameter,
2. Θ is injective.

Since $\tau \succeq \sigma$, there exists a substitution Φ with $\tau\Phi = \sigma$. Thus $\sigma\Theta\Phi = \sigma$. To show the first claim, for every type ϱ let $\mathtt{size}(\varrho)$ be the number of occurrences of type constructors in ϱ. For every parameter substitution Ψ, $\mathtt{size}(\varrho\Psi) \geq \mathtt{size}(\varrho)$. Further, if T is a parameter in ϱ and $T\Psi$ is not a parameter, then $\mathtt{size}(\varrho\Psi) > \mathtt{size}(\varrho)$.

Now assume, by contradiction, that some $T\Theta$ is not a parameter. Since T occurs in σ we have $\mathtt{size}(\sigma) = \mathtt{size}(\sigma\Theta\Phi) \geq \mathtt{size}(\sigma\Theta) > \mathtt{size}(\sigma)$ which is impossible.

We conclude the proof by showing that Θ does not map different parameters to the same parameter, i.e., we can not have $T\Theta = S\Theta$ if S is different from T. We argue as follows: Since for every parameter T we have that $T\Theta$ is a parameter, the number of parameters occurring in $\sigma\Theta$ is less or equal than the number of parameters in σ. Similarly, the number of parameters in $\sigma\Theta\Phi$ is less or equal than the number of parameters in $\sigma\Theta$. Assume that S and T both occur in σ and that $S\Theta = T\Theta$. But then $\sigma\Theta$ would contain strictly less parameters than σ. Therefore, $\sigma\Theta\Phi$ would also contain strictly less parameters than σ which is impossible since $\sigma = \sigma\Theta\Phi$. □

The Lemma just proved generalizes as follows.

Corollary 9 If $\langle \Gamma, \sigma\rangle \succeq \langle \Delta, \tau\rangle$ and $\langle \Delta, \tau\rangle \succeq \langle \Gamma, \sigma\rangle$, then there exists a renaming Θ of type parameters such that $\Gamma\Theta = \Delta$ and $\sigma\Theta = \tau$. ◇

The *uniqueness* of a principal type and an associated principal type assignment up to renaming is now immediate. The *existence* of a principal type and an associated principal type assignment is less obvious. The appendix presents an algorithm that, given a term t, checks whether t is typeable. Furthermore, in case t is typeable the algorithm computes a principal type of t and an associated principal type assignment.

In order to define the semantics of a typeable term t, we need to know the types of all subterms of t. Therefore, we define *annotated terms* next. Intuitively, an annotated term is a term where every subterm has been annotated with a type.

Definition 10 (Annotated Term) The notions of an *annotated term* α, the corresponding *naked* term $|\alpha|$, and the *type* of α are defined inductively:

1. If x is a variable and τ is a type, then $x : \tau$ is an annotated term of type τ. We have $|x : \tau| := x$.
2. If f is an n-ary function symbol, τ is a type and for all $i = 1, \ldots, n$ we have that α_i is an annotated term, then $f(\alpha_1, \ldots, \alpha_n) : \tau$ is an annotated term of type τ. Further, $|f(\alpha_1, \ldots, \alpha_n) : \tau| := f(|\alpha_1|, \ldots, |\alpha_n|)$. ◇

Like the definition of a term, the definition of an annotated term is purely syntactical and does not relate to the profiles of the functions symbols. We introduce the notion of an annotated term being *typeable* next.

Definition 11 (Typeable Annotated Term) If a type assignment Γ is given, then the notion of an annotated term α being entailed by Γ (denoted $\Gamma \vdash \alpha$) is defined inductively:

1. If $\Gamma(x) = \tau$, then $\Gamma \vdash x : \tau$.
2. If f is an n-ary function symbol, $\sigma_1 \times \ldots \times \sigma_n \to \tau$ is appropriate for f, $\Gamma \vdash \alpha_i$ for all $i = 1, \ldots, n$, and the type of α_i is σ_i, then $\Gamma \vdash f(\alpha_1, \ldots, \alpha_n) : \tau$.

An annotated term α is *typeable* if there exists a type assignment that entails it. ◇

If a typeable term t is given, then we can use the principal type assignment of t to compute an annotated term α such that $\Gamma \vdash \alpha$ and $|\alpha| = t$. This annotated term α is called the *full annotation* of t.

2.4 Quantifiers

According to the Lipari Guide [Gur95], terms may have quantifiers. For the simplicity of exposition, up to now we have considered only quantifier-free terms. Next, we extend the definition of terms and the related definitions to the general case.

First, extend the definition of terms to include quantifiers as well. That is, if t is a term, then $(\forall x)t$ and $(\exists x)t$ are terms. Free and bound variables of a term t are defined as expected. They are denoted by $\mathrm{FV}(t)$ and $\mathrm{BV}(t)$, respectively. It is convenient to assume that there are no name clashes between free and bound variables. Also, different occurences of quantifiers should bind different variables. This can always be achieved via a suitable variable renaming.

Remark: For applications and implementation, the following more explicit notation is convenient:

$$(\forall x : g)t, \qquad (\exists x : g)t.$$

Here g is a Boolan-valued term (the *guard*) containing the variable x and therefore limiting the quantification range. Logically these terms are equivalent to

$$(\forall x)(g \to t), \qquad (\exists x)(g \wedge t). \qquad \diamond$$

Next, we extend the definition of a term being *typeable*. To this end, we add the following clause to Definition 5.

3. If $\Gamma, x : \tau \vdash t : \mathtt{bool}$, then $\Gamma \vdash (\forall x)t : \mathtt{bool}$ and $\Gamma \vdash (\exists x)t : \mathtt{bool}$.

The definition of an *annotated term* is changed by admitting $\big((\forall x : \tau)t : \sigma\big) : \varrho$ and $\big((\exists x : \tau)t : \sigma\big) : \varrho$ as annotated terms. The definition of a *typeable annotated term* is then upgraded by adding the following clauses:

3. If $\Gamma, x : \tau \vdash \alpha$ and the type of α is $\mathtt{bool}$, then

$$\Gamma \vdash ((\forall x : \tau)\alpha) : \mathtt{bool} \quad \text{and} \quad \Gamma \vdash ((\exists x : \tau)\alpha) : \mathtt{bool}.$$

3 Basic Rules: Syntax and Semantics

In this section we turn to semantical notions. First, we introduce states. Essentially, a state is a many-sorted algebra, so there is not much difference between the notion of a state in the context of a typed ASM and the same notion as defined in the Lipari Guide [Gur95] for untyped ASMs. States are used to define the semantics of terms. Then, we proceed to give the syntax of *rules*. We conclude this section with the definition of their semantics.

3.1 States and the Semantics of Terms

Definition 12 (State) A *state* S is a pair $\langle [\![\cdot]\!], [\![\cdot, \cdot]\!] \rangle$ such that

1. $[\![\cdot]\!]$ is a function interpreting every closed type τ as a set $[\![\tau]\!]$.

2. $[\![\cdot,\cdot]\!]$ is a function interpreting every function symbol f together with any closed profile $\sigma_1 \times \ldots \times \sigma_n \to \tau$ appropriate for f as a function

$$[\![f_{\sigma_1 \times \ldots \times \sigma_n \to \tau}]\!] : [\![\sigma_1]\!] \times \ldots \times [\![\sigma_n]\!] \to [\![\sigma]\!].$$

A function of the form $[\![f_{\boldsymbol{\sigma \to \tau}}]\!]$ is called a *basic function* of the state S.

Furthermore, the functions $[\![\cdot]\!]$ and $[\![\cdot,\cdot]\!]$ have to satisfy the following restrictions:

1. $[\![\mathtt{bool}]\!] = \{\mathbf{true}, \mathbf{false}\}$ where **true** and **false** are distinct elements.
2. The interpretation of "**true**", "**false**", "$\neg$", "$\wedge$", "$\vee$", "$\to$", and "$=$" is as expected. Further, $[\![\mathbf{default_{bool}}]\!] = \mathbf{false}$. $\diamond$

With the definition of a state as it is given above it is not possible to evaluate terms containing quantifiers. In order to define the semantics of these terms we have to extend this definition. Assume a state $S = \langle [\![\cdot]\!], [\![\cdot,\cdot]\!] \rangle$ is given. Then we extend the vocabulary as follows: For every closed type τ and every element $c \in [\![\tau]\!]$ we choose a new nullary function symbol $\hat{c}$ with profile $\hat{c} : \tau$. The function $[\![\cdot,\cdot]\!]$ is extended to these new function symbols in the obvious way: we define $[\![\hat{c}_\tau]\!] := c$. In the following, if a state S is given, we will tacitly assume that the vocabulary is extended as described above and that the ensuing upgrading of the state is performed. Then, if t is a term possibly containing a variable x of type τ, τ is closed and $c \in [\![\tau]\!]$, then $t[x/\hat{c}]$ is the result of replacing the variable x in t with $\hat{c}$ everywhere. A similar notation is used for rules.

In general, a term t is evaluated bottom up so that all subterms of t are evaluated in the process. Call t *type-closed* if the principal type of t contains no parameters. Further, call t *hereditarily type-closed* if every subterm of t is type-closed. It is easy to see that the hereditary type-closedness is necessary and sufficient for t to be evaluable in all states. In a particular state, t can be evaluable even if this condition fails. Suppose, for example, that t is the term $\mathtt{length}(\mathtt{nil})$ where $\mathtt{length}$ is a function computing the length of a list. Although $\mathtt{nil}$ is not type-closed, the value of t is zero. We discuss this example in more detail below.

Next, we define the *value* of a typeable term t in a given state. We assume that t is closed, that is $\mathrm{FV}(t) = \emptyset$.

Definition 13 (Evaluation) Assume that a state $S = \langle [\![\cdot]\!], [\![\cdot,\cdot]\!] \rangle$ is given. First, we define the *value* of a typeable annotated term α that contains no type parameters. This value is denoted $[\![\alpha]\!]$. The definition of $[\![\alpha]\!]$ is by induction.

1. $[\![f(\alpha_1, \ldots, \alpha_n) : \tau]\!] := [\![f_{\sigma_1 \times \ldots \times \sigma_n \to \tau}]\!]([\![\alpha_1]\!], \ldots, [\![\alpha_n]\!])$,

   where σ_i is the type of α_i.

2. $[\![((\forall x : \sigma)t : \mathtt{bool}) : \mathtt{bool}]\!] :=$

$$\begin{cases} \mathbf{true} & \text{if } [\![t[x/\hat{c}] : \mathtt{bool}]\!] = \mathbf{true} \text{ for all } c \in [\![\sigma]\!]; \\ \mathbf{false} & \text{otherwise.} \end{cases}$$

3. $[\![((\exists x : \sigma)t : \mathtt{bool}) : \mathtt{bool}]\!] :=$

$$\begin{cases} \mathbf{true} & \text{if } [\![t[x/\hat{c}] : \mathtt{bool}]\!] = \mathbf{true} \text{ for at least one } c \in [\![\sigma]\!]; \\ \mathbf{false} & \text{otherwise.} \end{cases}$$

If t is hereditarily type-closed, let α be the full annotation of t. Clearly, α contains no type parameters. Define the *value* $[\![t]\!]$ of t equals $[\![\alpha]\!]$. $\diamond$

If a term t of principal type τ is given such that τ is closed, then t need not be hereditarily type closed, consider the term `length(nil)`. Since the profile of the function `length` is `list`$(T) \to$ `int`, the full annotation of `length(nil)` is given by `length(nil` : `list`(T)`)` : `int` and therefore `length(nil)` is not hereditarily type closed. The evaluation of such a term requires help from the user. For example, the user can annotate a function symbol with a type. In our example, an annotation `length(nil` : `int)` would solve the problem.

Fortunately, in practical applications of typed ASMs, hereditary type-closedness is rarely a problem. The reason is that terms of the form `length(nil)` do not appear in specifications of dynamic functions[3]. Instead, you may find something like `length(cons`$(c,$ `nil))`. However, if the type of c is τ and τ is closed, then the type of `nil` is constrained to be `list`(τ) and therefore the term `length`$(c,$ `nil)` is hereditarily type closed.

3.2 Syntax of Rules

We recall the definition of rules and give an intuitive but informal description of the semantics of the rules. At the same time, we define inductively the notion $\Gamma \vdash R$ where Γ is a type assignment. The notation $\Gamma \vdash R$ is read as "R is a well-typed rule w.r.t. Γ". A rule without free variables is *well-typed* if it is well type with respect to the empty type assignment.

1. *Skip Rule*: Let R be `skip`. Then R is a rule and $\Gamma \vdash$ `skip` for any type assignment Γ. The Skip Rule does contain neither free nor bound variables. Therefore, we have $\mathrm{FV}(R) := \mathrm{BV}(R) := \emptyset$.

   Intuitively, the Skip Rule does nothing.

2. *Update Rule*: Let R be

   $$f(s_1, \ldots, s_n) \texttt{ := } t,$$

   where s_1, ..., s_n, and t are terms. Then R is a rule and $\Gamma \vdash R$ iff there are types σ_i, ..., σ_n, and τ such that $\Gamma \vdash t : \tau$ and $\Gamma \vdash s_i : \sigma_i$ for all $i = 1, \ldots, n$. Furthermore, $\mathrm{FV}(R) := \mathrm{FV}(s_1) \cup \cdots \cup \mathrm{FV}(s_n) \cup \mathrm{FV}(t)$ and $\mathrm{BV}(R) := \mathrm{BV}(s_1) \cup \cdots \cup \mathrm{BV}(s_n) \cup \mathrm{BV}(t)$.

   Intuitively, the rule R updates the value of the dynamic functions f for the arguments $(s_1, \ldots, s_n)$ to t.

3. *Block Rule*: Let R be

   ```
   do in-parallel
         R1
         R2
   end-do.
   ```

[3] Of course, these terms do appear frequently in the specification of static functions as, e.g., `length`. However, the specification of static functions is not the topic of this paper.

Then R is a rule and $\Gamma \vdash R$ iff $\Gamma \vdash R_1$ and $\Gamma \vdash R_2$. Furthermore, we have $\mathrm{FV}(R) := \mathrm{FV}(R_1) \cup \mathrm{FV}(R_2)$ and $\mathrm{BV}(R) := \mathrm{BV}(R_1) \cup \mathrm{BV}(R_2)$.

Intuitively, R_1 and R_2 are executed in parallel.

4. *Conditional Rule*: Let R be

```
if  g
    then  R1
    else  R2
end-if.
```

Then R is a rule and $\Gamma \vdash R$ iff $\Gamma \vdash R_1$ and $\Gamma \vdash R_2$. Furthermore, we have $\mathrm{FV}(R) := \mathrm{FV}(R_1) \cup \mathrm{FV}(R_2)$ and $\mathrm{BV}(R) := \mathrm{BV}(R_1) \cup \mathrm{BV}(R_2)$.

Intuitively, execution of R means execution of R_1 if g is true and execution of R_2 otherwise.

5. *Do-forall Rule*: Let R be

```
do forall x satisfying g
      R0
end-do.
```

Then R is a rule and $\Gamma \vdash R$ iff, first, $x \notin \mathbf{dom}(\Gamma)$ and, second, there is a type τ such that $\Gamma, x : \tau \vdash g :$ `bool` and $\Gamma, x : \tau \vdash R_0$. The variable x is bound in R. Therefore, we have $\mathrm{FV}(R) := \big(\mathrm{FV}(g) \cup \mathrm{FV}(R_0)\big) - \{x\}$ and $\mathrm{BV}(R) := \mathrm{BV}(R_0) \cup \{x\}$.

Intuitively, execution of R means executing in parallel all rules $R_0[x/\hat{c}]$ for which the condition $g[x/\hat{c}]$ is satisfied.

6. *Choose Rule*: Let R be

```
choose x satisfying g
      R0
end-choose.
```

Then R is a rule and $\Gamma \vdash R$ iff, first, $x \notin \mathbf{dom}(\Gamma)$ and, second, there is a type τ such that $\Gamma, x : \tau \vdash g :$ `bool` and $\Gamma, x : \tau \vdash R_0$. The variable x is bound in R. Therefore, we have $\mathrm{FV}(R) := \big(\mathrm{FV}(g) \cup \mathrm{FV}(R_0)\big) - \{x\}$ and $\mathrm{BV}(R) := \mathrm{BV}(R_0) \cup \{x\}$.

Intuitively, executing R amounts to choosing an element c such that $g[x/\hat{c}]$ holds and then executing $R_0[x/\hat{c}]$.

A program is just a rule. A program without any choose rules is called *deterministic*. In the following, it is convenient to distinguish between *static* and *dynamic* functions. A function f is called *static* (with respect to a given program) if the program contains no updates of the form $f(s_1, \ldots, s_n) := t$. Obviously, a static function does not change during the computation[4].

[4] In this paper, we do not discuss the notion of external functions, i.e. functions that are changed by the environment. If we admit external functions as discussed in the Lipari Guide, then the definition of a static function is changed. We then have to require additionally that a static function is not updated by the environment.

Example 7 Assume we have a type `Vertex` representing nodes of a colored graph. Assume further that l is a nullary dynamic function representing a list of nodes and

$$\texttt{color} : \texttt{Vertex} \to \texttt{Color}$$

is the function yielding the color of a vertex. We suppose that one of the colors is `red`. If $\Gamma = \{x : \texttt{Vertex}\}$ then

$$\Gamma \vdash \texttt{color}(x) := \texttt{red}$$

To proceed, assume the static function

$$\texttt{member} : T \times \texttt{list}(T) \to \texttt{bool}$$

is given and has the obvious meaning. Then, if R is

```
do forall x satisfying member(x, l)
   color(x) := red
end-do,
```

then R is a well-typed rule, since we have $\emptyset \vdash R$. Intuitively, execution of R colors all nodes in l with `red`. Note that R does not contain free variables. ◇

A rule is called *transparent* iff there are no name clashes between two bound variables or between a bound variable and a free variable. As a matter of convenience, we will use only transparent rules, renaming variables if necessary. This assumption is convenient in order to define the semantics of rules that bind variables, i.e., the Do-forall Rule and the Choose Rule. In defining the semantics of these rules, we substitute constants for variables. These substitutions would be much more cumbersome to define if they had to be aware of free and bound variables.

3.3 Semantics of Deterministic Rules

The semantics of rules is given by sets of *updates*.

Definition 14 (Update) A triple $\langle f, \langle x_1, \ldots, x_n \rangle, y \rangle$ is an *update* iff f is an n–ary function symbol and there exists a profile $\sigma_1 \times \ldots \times \sigma_n \to \tau$ appropriate for f such that

1. $\mathrm{Par}(\tau) = \emptyset$ and $\mathrm{Par}(\sigma_i) = \emptyset$ for all $i = 1, \ldots, n$,
2. $y \in [\![\tau]\!]$ and $x_i \in [\![\sigma_i]\!]$ for all $i = 1, \ldots, n$. ◇

Conceptually, an update specifies how the function table of a dynamic function has to be *updated*. An *update set* is a set of updates. In order to define the semantics of a well-typed rule, we have to extend the notion of hereditary type closedness to rules. Define the *full annotation* of a rule R by replacing all terms in R by their full annotation. Call a rule R *executable* iff, first, its set of free variables is empty and, second, its full annotation is hereditarily type closed, i.e., iff no type parameters occur in its full annotation. Then, we define the semantics of an executable rule R in state S as an update set. This update set is denoted by $\texttt{den}(R, S)$. It is defined by induction:

1. R is `skip`. Then

$$\mathtt{den}(R,S) \;:=\; \emptyset.$$

2. R is $f(s_1,\ldots,s_n) \;:=\; t$. Then

$$\mathtt{den}(R,S) \;:=\; \big\{\big\langle f, \langle [\![s_1]\!],\ldots,[\![s_n]\!]\rangle, [\![t]\!]\big\rangle\big\}.$$

3. R is

```
do in-parallel
      R1
      R2
end-do.
```

   Then

$$\mathtt{den}(R,S) \;:=\; \mathtt{den}(R_1,S) \cup \mathtt{den}(R_2,S).$$

4. R is

```
if  g
    then  R1
    else  R2
end-if.
```

   Then

$$\mathtt{den}(R,S) = \begin{cases} \mathtt{den}(R_1,S) & \text{if } [\![g]\!] = \mathbf{true};\\ \mathtt{den}(R_2,S) & \text{if } [\![g]\!] = \mathbf{false}. \end{cases}$$

5. R is

```
do forall x satisfying g
      R0
end-do.
```

   Then there is a type assignment Γ and a type τ such that $\Gamma, x:\tau \vdash R_0$. Define

$$\mathtt{den}(R,S) \;:=\; \bigcup\big\{\mathtt{den}(R_0[x/\hat{c}],S):\; c \in [\![\tau]\!] \wedge [\![g[x/\hat{c}]]\!] = \mathbf{true}\big\}.$$

Finally, to *fire* a rule R in a state S compute the update set $\mathtt{den}(R,S)$. Then change the state S by updating the values of the functions $[\![f_{\sigma_1\times\ldots\times\sigma_n\to\tau}]\!]$ as prescribed by the update set $\mathtt{den}(R,S)$, i.e., for every update $\big\langle f, \langle x_1,\ldots,x_n\rangle, y\big\rangle$ in $\mathtt{den}(R,S)$ the interpretation of f is changed so that

$$[\![f_{\sigma_1\times\ldots\times\sigma_n\to\tau}]\!](x_1,\ldots,x_n) = y$$

holds.

Remark: We saw above that a term can be evaluable in some states even if it is not hereditary type-closed. A similar observation applies to rules. Our notion of executability of a rule R guarantees that R is executable in all states (of sufficiently rich vocabulary). A rule that is not hereditary type-closed can be executable in some states.

3.4 Semantics of Non-deterministic Rules

We first give the informal semantics of the Choose Rule. Assume that the rule R has the form

```
choose x satisfying g
```

R_0
end-choose.

and let S be a state. Then there is a type assignment Γ and a type τ such that $\Gamma, x : \tau \vdash R_0$. To fire R at S, check if the set

$$\{c \in [\![\tau]\!] \ : \ [\![g[x/\hat{c}]]\!] = \mathbf{true}\}$$

is empty. If it is, do nothing. Otherwise, choose any element c from this set and fire the rule $R_0[x/\hat{c}]$.

In order to give the formal semantics for non-deterministic rules one has to define the notion of the non-deterministic semantics $\mathtt{nden}(R, S)$ of a rule R in a state S as a family of update sets. Firing a rule is then done by non-deterministically choosing an update set from this family and then firing this update set. Here, we forgo a formal definition and refer to [Gur97] for the details. It is straightforward to adapt the definition given in the above reference to the typed case.

4 An Example

A specification language that is easy to reason about should be concise. But one needs syntactic sugar to ease the work of specification writing. For example, there are some pretty obvious generalizations of the syntax for rules given in the previous sections. To give an example, the construction

if g **then** R **end-if**

is considered to be an abbreviation of

if g **then** R **else skip end-if**.

In a similar way, the construction

let $x = t$ **in** R **end-let**

abbreviates $R[x/t]$. This construction is useful if t is a complex term that has serveral occurrences in the rule R. Furthermore, the Block Rule is generalized to contain any finite set of rules and the Do-forall Rule is generalized to support quantification over a finite number of variables. For reasons of space we do not give the details.

In the following example, we make heavy use of the type `list`. This type is constructed by the two function symbols `nil` and `cons`. Their profiles have already been given in Section 2, we repeat them here for the reader's convenience:

$$\mathtt{nil} \ : \mathtt{list}(T)$$
$$\mathtt{cons} : T \times \mathtt{list}(T) \to \mathtt{list}(T)$$

To simplify notation, we adopt the following conventions for writing terms of type `list`: We write `[]` instead of `nil` and $[x \,|\, xs]$ instead of $\mathtt{cons}(x, xs)$. Furthermore, if l is a given list, x is variable and t is a term of type `bool` containing the variable x, then $[\, x \in l \,|\, t \,]$ is the list of those elements of l which satisfy the condition t.

We need some static functions for lists. First, there is the function `append`

$$\mathtt{append} : \mathtt{list}(T) \times \mathtt{list}(T) \to \mathtt{list}(T).$$

This function is defined by the following recursive equations.

$$\texttt{append}([\,], l) = l$$
$$\texttt{append}([\,x \mid l_1\,], l_2) = [\,x \mid \texttt{append}(l_1, l_2)\,].$$

Further, the function

$$\texttt{member} : T \times \texttt{list}(T) \to \texttt{bool}$$

checks whether the first argument is an element of the second argument. Instead of $\texttt{member}(x, l)$ we use the shorthand $x \in l$.

Next, we consider the Graph Reachability problem:

Given: A graph $G = (V, E)$ with distinguished nodes `source` and `target`.

Question: Does there exists a path from `source` to `target` in G?

Before we give an algorithm for this problem, we discuss the representation of the graph. We assume that `Vertex` is a nullary type constructor that is interpreted as the set of vertices. The set of vertices V of the given graph is represented by the nullary function `nodes` with profile

$$\texttt{nodes} : \texttt{list}(\texttt{Vertex}).$$

In our modelling, we represent the set of edges E by the binary relation

$$\texttt{edge} : \texttt{Vertex} \times \texttt{Vertex} \to \texttt{bool}.$$

A common algorithm that solves the Graph Reachability problem for graphs that contain no cycles proceeds by iteratively constructing a list `reachable` of vertices reachable from `source`. To compute `reachable`, an auxiliary "border list" `border` that is a sublist of `reachable` is used. Therefore, `reachable` and `border` are dynamic functions with the following profiles:

$$\texttt{reachable} : \texttt{list}(\texttt{Vertex})$$
$$\texttt{border} : \texttt{list}(\texttt{Vertex})$$

We need a dynamic function guiding the flow of control. Its profile is

$$\texttt{mode} : \texttt{Mode},$$

where the nullary type constructor `Mode` represents a set containing the strings `initial`, `construct`, and `examine`. Finally, as a result of the computation, the dynamic function

$$\texttt{output} : \texttt{bool}$$

will answer the question whether there is a path from `source` to `target`.

The algorithm is now implemented by three rules. The first rule initializes the lists `source` and `border` to the singleton list [`source`].

```
if  mode = initial
    then do in-parallel
            reachable := [source]
            border := [source]
            mode := construct
         end-do
end-if
```

The next rule is the working horse of the algorithm. It iteratively extends the list of reachable nodes:

```
if  mode = construct
    then if   border ≠ []
         then choose x satisfying x ∈ border
                 let l = [ y ∈ nodes | edge(x, y) ∧ y ∉ border) ] in
                     do in-parallel
                         reachable := append(reachable, l)
                         border := append([ z ∈ border | z ≠ x ], l)
                     end-do
                 end-let
              end-do
         else mode := examine
         end-if
end-if
```

The final rule checks whether `target` is an element of the list `reachable`.

```
       if  mode = examine
           then if   target ∈ reachable
                then output := true
                else output := false
                endif
       endif
```

The rules given above are well-typed. This can be checked mechanically using a type checker. For example, we have used the type checker that has been implemented as part of a *workbench* for ASMs developed at the University of Paderborn [Cas98]. Of course, the benefit of type checking gets most obvious when dealing with ill-typed terms. Therefore, let us assume that the second rule is changed as follows:

```
if  mode = construct
    then if   border ≠ []
         then choose x satisfying x ∈ border
                 let l = [ y ∈ nodes | edge(x, y) ∧ y ∉ border) ] in
                     do in-parallel
                         reachable := append([reachable], l)
                         border := append([ z ∈ border | z ≠ x ], l)
                     end-do
                 end-let
              end-do
         else mode := examine
         end-if
end-if
```

Above, we have changed the update

```
reachable := append(reachable, l)
```

into

```
reachable := append([reachable], l)
```

The resulting rule is no longer well-typed. The type checker produces the error message

```
type check error -- function
    append: list(T) * list(T) -> list(T)
called with argument of type
            list(list(Vertex)) * list(Vertex)
```

(though in a slightly different syntax). The error message is due to the fact that `append` expects its arguments to be lists of the same type, but in the erroneous update the first argument of `append` has type `list(list(Vertex))`, while the second argument has type `list(Vertex)`.

Of course, the previous example is rather small and therefore the error is pretty obvious. However, the situation changes when the specifications get larger. When rules are changed in the maintenance phase of a big specification, people maintaining this specification tend to forget dependencies between different parts of it. Often, type errors are the result. Then a type checker is a valuable tools that helps keeping the specification consistent.

5 Concluding Remarks

When one defines a new concept, there often is a choice between different alternatives. In order to choose between these alternatives, one needs some guiding principles. In the first subsection, we formulate some principles and discuss how they have influenced the design of typed ASMs. The second subsection is devoted to the issue of polymorphic dynamic functions. Finally, in the third subsection, we discuss our motivation for excluding the import construct from typed ASMs.

5.1 Guiding Principles

First and foremost, a specification language has to be simple. A customer that needs a new software or hardware system does not usually specify his system rigorously. Instead, he hires a requirement analyst. Together, they create an initial specification. It is important that the customer is able to read this specification in order to assess whether the system meets his requirements. This calls for the specification language to be simple. But then the type system of this specification language should be simple too. In this connection, we opted to adapt the type system of ML to our needs. That type system is both simple and well understood.

Of course, simplicity is only one requirement on a type system. Another is expressibility. There is a tension between the two requirements. For example, a satisfactory integration of parametric polymorphism (which plays an important role in functional programming) and inclusion polymorphism (which plays an important role in the object oriented approach) is an open problem, and the type system of ML lacks inclusion polymorphism. In [GS98], we will extend our type system in the direction of inclusion polymorphism along the lines of [Bei95b].

One of the important features of the ASM approach is that specifications can be given at any level of abstraction. To keep this freedom for typed ASMs,

we have left the interpretation of types abstract. The only requirement is that every closed type is interpreted as a set. We could have made a different choice by requiring the type constructors to be interpreted as functions taking sets as inputs and producing sets as outputs. But nothing would be gained in the alternative approach unless we specify how the type constructors work. However, if we specify how the constructors work, the freedom of abstraction would be lost.

Therefore, we leave the specification of type constructors open. Of course, the user of typed ASMs is free to be more specific. In a number of applications it is convenient to specify type constructors via certain constructor functions that generate the type in question as a term algebra modulo some set of axioms. To give an example, the unary type constructor `Set` denoting finite sets is conveniently defined as the algebra generated by the constructor functions

$$\begin{aligned}&\texttt{empty} : \texttt{Set}(T),\\ &\texttt{insert} : T \times \texttt{Set}(T) \to \texttt{Set}(T),\end{aligned}$$

satisfying the equations

$$\begin{aligned}&\texttt{insert}\big(x, \texttt{insert}(y, s)\big) = \texttt{insert}\big(y, \texttt{insert}(x, s)\big),\\ &\texttt{insert}(x, \texttt{insert}(x, s)) = \texttt{insert}(x, s).\end{aligned}$$

With the current definition of typed ASMs the user has the freedom to use definitions of this (or other) kind. But we do not want to confine the user to the algebraic style of defining data types. This is the main difference between our approach and the work of Zamulin [Zam97]. Zamulin offers a system that is not confined to the specification of the dynamic parts of an algorithm. He also introduces a specification language for specifying the static part of an ASM. This provides obvious benefits but much of the flexibility of the ASM approach is lost in the process. With untyped ASMs, the user can use any method for specifying the static part. The present paper preserves this freedom. This freedom is essential for the integration of formal methods into an industrial development process. Often, engineers already use semiformal specification techniques for parts of their design. If a formal specification language is to be successfully integrated into an industrial process, it is vital that the language can incoporate as much as possible of the existing notation. This is our reason for not prescribing how the static part of an ASM should be specified.

5.2 A Remark on Polymorphic Dynamic Functions

Recall that a term t is evaluable in all states only if it is hereditarily type-closed. In this connection, one may want to require that the profiles of dynamic functions contain no type parameters. This strong condition still allows for a kind of implicit polymorphism. To see this, consider the problem of finding the shortest path in a graph. In general, an algorithm for computing the shortest path does not need to know the nature of the vertices of the graph. It is no problem to formulate a general algorithm of this kind as an ASM [Str97]. The idea is to have a nullary type constructor `Vertex` whose interpretation is left abstract. Using this abstract type, we can formulate an algorithm that is essentially polymorphic in the type `Vertex`, although formally `Vertex` is not a type parameter.

One may introduce the notion of a *procedure* so that every procedure P defines a number of type parameters that remain fixed during the execution of

P. Then one can write, for example, a procedure P for computing the shortest path in a graph. `Vertex` would be a type parameter that is fixed in P and can appear, without endangering the evaluability of terms, in function profiles of dynamic functions specific to P.

Without introducing procedures (as in this paper), one can use a nullary type constructor to play the role of a type parameter which would be fixed in a procedure.

5.3 Import and Reserve

One major difference between typed and untyped ASMs is the absence of the import rules in typed ASMs. To recall, in untyped ASMs, an import rule

```
import x
       R_0(x)
end-import
```

chooses an element a from the so called *reserve* and executes the rule $R_0(a)$. The reserve consists of indistinguishable elements which do not belong to any particular universe. No function produces a reserve element, and every function produces a default value if at least one of the arguments is in the reserve. The idea of the reserve does not fit well the static and structured paradigm of types. To illustrate one difficulty, consider for example the type $\texttt{list}(T)$ introduced previously. The value of the term $\texttt{cons}(r, [])$ should be undefined if r is a reserve element; it becomes defined when r is imported. But then `cons` is no longer a static function. Thus we have decided to drop the *import* construct. Import was convenient to extend, for example, the current set of nodes. There are alternative ways to achieve the same goals.

Dropping the reserve renders a number of definitions simpler. For example, the *May 1997 Draft of the ASM Guide* [Gur97] defines the denotation of a deterministic rule as an equivalence class of update sets: two update sets that differ merely in the choice of reserve elements are equivalent. Here, we do not have reserve elements around and can define the denotation of a deterministic rule as an update set.

References

[ASM] Abstract state machines. Web site of the University of Michigan at `http://www.eecs.umich.edu/gasm/`.

[Bei95a] Christoph Beierle. Concepts, implementation, and applications of a typed logic programming language. In Beierle and Plümer [BP95], chapter 5, pages 139–167.

[Bei95b] Christoph Beierle. Type inferencing for polymorphic order-sorted logic programs. In Leon Sterling, editor, *Proceedings of the 1995 International Conference on Logic Programming*. MIT Press, 1995.

[BP95] Christoph Beierle and Lutz Plümer, editors. *Logic Programming: Formal Methods and Practical Applications*. Studies in Computer Science and Artificial Intelligence. Elsevier Science B.V./North-Holland, Amsterdam, Holland, 1995.

[Cas98] Giuseppe Del Castillo. ASM-SL, a specification language based on Gurevich's Abstract State Machines: Introduction and tutorial. Technical report, Universität-GH Paderborn, 1998. to appear.

[Fit95] Melvin Fitting. *First-order logic and automated theorem proving.* Texts and monographs in computer science. Springer, New York, second edition, 1995.

[FM90] You-Chin Fuh and Prateek Mishra. Type inference with subtypes. *Theoretical Computer Science*, 73(2):155–175, 1990.

[GS98] Yuri Gurevich and Karl Stroetmann. Typed ASMs: Adding casting and overloading (*tentative title*), 1998. In preparation.

[Gur95] Yuri Gurevich. Evolving algebras 1993: Lipari guide. In Egon Börger, editor, *Specification and Validation Methods*, pages 3–36. Oxford University Press, 1995. Available at [ASM].

[Gur97] Yuri Gurevich. May 1997 draft of the ASM guide. Technical Report CSE-TR-336-97, University of Michigan, EECS Department, 1997. Available at [ASM].

[Hin96] Roger Hindley. *Basic Simple Type Theory.* Cambridge University Press, 1996.

[HL92] Patricia M. Hill and John W. Lloyd. The Gödel programming language. Technical Report CSTR–92–27, Department of Computer Science, University of Bristol, 1992.

[HSC96] Fergus Henderson, Zoltan Somogyi, and Thomas Conway. Determinism analysis in the MERCURY compiler. In *Proceedings of the Australian Computer Science Conference*, pages 337–346, Melbourne, Australia, January 1996.

[LP97] Leslie Lamport and Lawrence C. Paulson. Should your specification language be typed. Report 147, DEC Systems Research Center, Palo Alto, CA, May 1997.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 1978.

[Mit96] John C. Mitchell. *Foundations for Programming Languages.* The MIT Press, 1996.

[MM82] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Language Systems*, 4:258–282, 1982.

[Mor68] Jim H. Morris. *Lambda Calculus Models of Programming Languages.* PhD thesis, MIT, 1968.

[MTH90] Robin Milner, Mats Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

[NHN80] R. Nakajima, M. Honda, and H. Nakahara. Hierarchical program specification: a many-sorted logical approach. *Acta Informatica*, 14:135–155, 1980.

[Rob65] John Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[Str97] Karl Stroetmann. The constrained shortest path problem: A case study in using ASMs. *J.UCS*, 3(4):304–319, 1997.

[Wan87] Mitchell Wand. A simple algorithm and proof for type inference. *Fundamenta Infomaticae*, 10:115–122, 1987.

[WB89] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, 1989.

[Zam97] Alexandre V. Zamulin. Typed Gurevich machines revisited. *Joint NCC and IIS Bulletin of Computer Science*, 5:1–30, 1997.

Acknowledgements

We thank Egon Börger, Bertil A. Brandin, Sabine Glesner, Jim Huggins, Martin Müller, Peter Päppinghaus, and Alexandre V. Zamulin for commenting on drafts of this paper. Furthermore, we would like to thank three anonymous referees, who have given very detailed and helpful comments.

The second author has been partially supported by NSF grant CCR 95-04375 and ONR grant N00014-94-1-1137. The third author has been partially supported by BMBF grant 01 IS 519 A 9.

A Type Inference

In the following we need the notion of a *type equation*. This is a string of the form $\sigma \simeq \tau$ where σ and τ are types. A parameter substitution Φ *solves* the type equation $\sigma \simeq \tau$ iff $\sigma\Phi = \tau\Phi$.

The type inference algorithm presented below proceeds in two stages: First, we compute a set of type equations. Secondly, we decide whether these type equations are solvable using the Martelli-Montanari algorithm for unification [MM82]. Our algorithm for type inference is derived from the algorithm presented by [Wan87] for type checking lambda terms.

We proceed to give the details of our algorithm. The purpose of this algorithm is to check whether a term t is typeable and to compute a principal type and an associated principal type assignment in case t is typeable. To this end we need to find a type assignment Γ and a type τ such that $\Gamma \vdash t : \tau$ can be established. We proceed as follows:

1. We associate a unique type parameter T_x with every variable x in such a way that the type parameters associated with different variables are distinct. It is assumed that there are infinitely many type parameters that are not of the form T_x. We call parameters of the form T_x *variable-linked*.
2. We define a universal type assignment Δ by setting $\Delta(x) := T_x$. For the rest of this paper, Δ is assumed to denote this particular type assignment.
3. We define a function `TypeEqs` that takes a term t and a type π and produces a set of type equations. The definition of the function `TypeEqs` will ensure that t is typeable iff $\texttt{TypeEqs}(t : T)$ is solvable. Here T is any *fresh* type parameter, that is a type parameter that is not variable-linked and does not occur in the profiles of the function symbols in t. Furthermore, if Φ is a *most general solution* of $\texttt{TypeEqs}(t : T)$, then $T\Phi$ is a principal type of t and $\Delta\Phi$ is the associated principal type assignment.
4. The question whether $\texttt{TypeEqs}(t : T)$ is solvable is decided by the Martelli-Montanari unification algorithm [MM82]. This algorithm also computes a *most general solution* in case $\texttt{TypeEqs}(t : T)$ is solvable. Unification is discussed in a number of text books on logic, e.g. [Fit95].

Now all we have to do is to define the function `TypeEqs` and to prove that it has the desired properties. We give a simple inductive definition of $\texttt{TypeEqs}(t : \pi)$. The basic idea is to apply the inductive definition of typeable terms backwards.

Definition 15 (`TypeEqs`)

1. If x is a variable, then
$$\texttt{TypeEqs}(x : \pi) := \{\pi \simeq T_x\}.$$
2. Assume $f : \sigma_1 \times \ldots \times \sigma_n \to \tau$. Then
$$\texttt{TypeEqs}\big(f(s_1, \ldots, s_n) : \pi\big) := \{\pi \simeq \tau\} \cup \bigcup_{i=1}^{n} \texttt{TypeEqs}(s_i : \sigma_i).$$
3. $\texttt{TypeEqs}\big((\forall x)t : \pi\big) := \big\{\pi \simeq \texttt{bool}\big\} \cup \texttt{TypeEqs}(t : \texttt{bool})$.
4. $\texttt{TypeEqs}\big((\exists x)t : \pi\big) := \big\{\pi \simeq \texttt{bool}\big\} \cup \texttt{TypeEqs}(t : \texttt{bool})$. $\diamond$

The definition of the type inference algorithm is now complete. Of course, we still have to prove its correctness. This is done in the following subsection. To simplify this exposition, we restrict our attention to the case of terms containing no quantifiers.

Note: The type inference algorithm presented above infers the types of all variables. Sometimes the user may want to declare the types of certain variables. The type inference algorithm is easily adapted to provide this additional flexibility: If the user declares that the variable x occurring in a term t has type τ, then the equation $T_x = \tau$ is added to the set of type equations generated for checking whether t is typeable. $\diamond$

Example 8 Assume that t is $\mathtt{append}(\mathtt{cons}(x, xs), ys)$. Assume further that the profiles of the function symbols occurring in t are given as follows:

$$\begin{array}{l}\mathtt{cons}: R \times \mathtt{list}(R) \to \mathtt{list}(R),\\ \mathtt{append}: \mathtt{list}(S) \times \mathtt{list}(S) \to \mathtt{list}(S).\end{array}$$

Then $\mathtt{TypeEqs}(t : T)$ is calculated to be the following set:

$$\{\ T \simeq \mathtt{list}(S), \mathtt{list}(S) \simeq \mathtt{list}(R), R \simeq T_x, \mathtt{list}(R) \simeq T_{xs}, \mathtt{list}(S) \simeq T_{ys}\ \}$$

It is easy to see that the substitution Θ defined as

$$[\ T \mapsto \mathtt{list}(S),\ R \mapsto S,\ T_x \mapsto S,\ T_{xs} \mapsto \mathtt{list}(S),\ T_{ys} \mapsto \mathtt{list}(S)\ ]$$

is a most general solution of $\mathtt{TypeEqs}(t : T)$. This shows that t is typeable and that $\mathtt{list}(S)$ is a principal type of t. A principal type assignment is the following:

$$\{\ x : S,\ xs : \mathtt{list}(S),\ ys : \mathtt{list}(S)\ \}.$$ $\diamond$

A.1 Correctness of the Type Inference Algorithm

Recall that a *type annotation* is a pair $t : \tau$. A parameter substitution Φ *solves* a type annotation iff $\Delta\Phi \vdash t : \tau\Phi$ holds. A *type constraint* is either a type equation or a type annotation. A parameter substitution Φ solves a set of type constraints C iff it solves every type equation and every type annotation in C. This is written $\Phi \models C$. We define a rewrite relation on sets of type constraints. It is the least transitive relation $\rightsquigarrow$ such that:

1. $C \cup \{x : \pi\} \rightsquigarrow C \cup \{\pi \simeq T_x\}$.
2. Assume that $f : \sigma_1 \times \ldots \times \sigma_n \to \tau$. Then
$$C \cup \{f(s_1, \ldots, s_n) : \pi\} \rightsquigarrow C \cup \{\pi \simeq \tau\} \cup \{s_i : \sigma_i \mid i = 1, \ldots, n\}.$$

If an annotation $t : \pi$ is given, then the two rewrite rules can be used repeatedly until the set $\mathtt{TypeEqs}(t : \pi)$ is derived. This is easily seen by induction on t. Furthermore, the rewrite relation $\rightsquigarrow$ satisfies the following invariants.

1. $\Phi \models C_2 \ \wedge\ C_1 \rightsquigarrow C_2 \ \Rightarrow\ \Phi \models C_1$ (I1)
2. $\Phi \models C_1 \ \wedge\ C_1 \rightsquigarrow C_2 \ \Rightarrow\ \exists\Psi \cdot (\Phi \subseteq \Psi \ \wedge\ \Psi \models C_2)$ (I2)

Before proving these invariants, let us observe that they suffice to reach our goals which we formulate as Soundness Theorem and Completeness Theorem.

Theorem 16 (Soundness) If Φ solves $\mathtt{TypeEqs}(t : \pi)$, then $\Delta\Phi \vdash t : \pi\Phi$.

Proof: By the assumption, $\Phi \models \texttt{TypeEqs}(t : \pi)$. Since the constraint set $\{t : \pi\}$ rewrites to $\texttt{TypeEqs}(t : \pi)$, the invariant (I1) shows $\Phi \models t : \pi$. □

Theorem 17 (Completeness) If $\Gamma \vdash t : \pi$ and T_0 is a fresh type parameter (that is neither variable-linked nor occurs in Γ or π), then $\texttt{TypeEqs}(t : T_0)$ is solvable. If Ψ is a most general solution of $\texttt{TypeEqs}(t : T_0)$, then $T_0\Psi$ is a principal type of t with associated principal type assignment $\Delta\Psi$.

Proof: Assume $\Gamma \vdash t : \pi$. W.l.o.g. $\mathbf{dom}(\Gamma) \subseteq \mathrm{FV}(t)$. If $\mathrm{FV}(t) = \{x_1, \ldots, x_n\}$, define the parameter substitution Φ as

$$\Phi \;:=\; [T_0 \mapsto \pi, T_{x_1} \mapsto \Gamma(x_1), \ldots, T_{x_n} \mapsto \Gamma(x_n)].$$

Then, $\Phi \models t : T_0$. Since $\texttt{TypeEqs}(t : T_0)$ rewrites to $\{t : T_0\}$, the invariant (I1) shows that Φ can be extended to a parameter substitution Θ such that $\Theta \models \texttt{TypeEqs}(t : T_0)$. Therefore $\texttt{TypeEqs}(t : T_0)$ is solvable. Next, assume Ψ is the most general solution. Then Θ can be written as $\Psi\Lambda$ for an appropriate parameter substitution Λ. Since $\pi = T_0\Phi = T_0\Theta = (T_0\Psi)\Lambda$ and $\Gamma = \Delta\Phi = \Delta\Theta = (\Delta\Psi)\Lambda$ the claim is established. □

Proof of (I1): According to the definition of the rewrite relation $\rightsquigarrow$, it suffices to consider the following two cases.

1. $C_1 = C \cup \{x : \pi\} \rightsquigarrow C \cup \{\pi \simeq T_x\} = C_2$. The assumption is that $\Phi \models C_2$. Then $\Phi \models C$ and $T_x\Phi = \pi\Phi$. Therefore $\Delta\Phi \vdash x : \pi\Phi$ and $\Phi \models C_1$.
2. $C_1 = C \cup \{f(s_1, \ldots, s_n) : \pi\} \rightsquigarrow C \cup \{\pi \simeq \tau\} \cup \{s_i : \sigma_i \mid i = 1, \ldots, n\} = C_2$, where the profile of f is given as $f : \sigma_1 \times \ldots \times \sigma_n \to \tau$. According to our assumption we have $\Phi \models C$, $\pi\Phi = \tau\Phi$, and $\Phi \models s_i : \sigma_i$ for $i = 1, \ldots, n$. Then $\Delta\Phi \vdash s_i : \sigma_i\Phi$. Therefore, $\Delta\Phi \vdash f(s_1, \ldots, s_n) : \tau\Phi$ and that yields the claim. □

Proof of (I2): Again, it suffices to consider the two cases corresponding to the two rewrite rules.

1. $C_1 = C \cup \{x : \pi\} \rightsquigarrow C \cup \{\pi \simeq T_x\} = C_2$. The assumption is that $\Phi \models C_1$. Then $\Phi \models C$ and $\Delta\Phi \vdash x : \pi\Phi$. Therefore $T_x\Phi = \pi\Phi$. Define $\Psi := \Phi$.
2. $C_1 = C \cup \{f(s_1, \ldots, s_n) : \pi\} \rightsquigarrow C \cup \{\pi \simeq \tau\} \cup \{s_i : \sigma_i \mid i = 1, \ldots, n\} = C_2$, where the profile of f is given as $f : \sigma_1 \times \ldots \times \sigma_n \to \tau$. Without loss of generality, we may assume that the type parameters that occur in this profile do not occur in $\mathbf{dom}(\Phi)$; the type parameters in the profile can be renamed. According to our assumption we have $\Phi \models C$ and $\Phi \models f(s_1, \ldots, s_n) : \pi$. The latter implies $\Delta\Phi \vdash f(s_1, \ldots, s_n) : \pi\Phi$. Lemma 6 shows that there is a parameter substitution Θ such that

   $$\Delta\Phi \vdash s_i : \sigma_i\Theta \quad \text{for all } i = 1, \ldots, n$$

   and $\pi\Phi = \tau\Theta$. We can assume that $\mathbf{dom}(\Theta)$ contains only type parameters occurring in the profile of f. Then $\mathbf{dom}(\Phi)$ and $\mathbf{dom}(\Theta)$ are disjoint. Define $\Psi := \Phi \cup \Theta$. □