BRICS RS-04-35 J. Iversen: Type Checking Semantic Functions in ASDF

BRICS

Basic Research in Computer Science

Type Checking Semantic Functions in ASDF

Jørgen Iversen

BRICS Report Series **RS-04-35**

ISSN 0909-0878 **December 2004**

**BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark
Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk**

Type checking semantic functions in ASDF

Jørgen Iversen
BRICS & Department of Computer Science
University of Aarhus
IT-parken, Aabogade 34
DK-8200 Aarhus N, Denmark
j.iversen@brics.dk

Abstract

When writing semantic descriptions of programming languages, it is convenient to have tools for checking the descriptions. With frameworks that use inductively defined semantic functions to map programs to their denotations, we would like to check that the semantic functions result in denotations with certain properties. In this paper we present a type system for a modular style of the action semantic framework that, given signatures of all the semantic functions used in a semantic equation defining a semantic function, performs a soft type check on the action in the semantic equation.

We introduce types for actions that describe different properties of the actions, like the type of data they expect and produce, whether they can fail or have side effects, etc. A type system for actions which uses these new action types is presented. Using the new action types in the signatures of semantic functions, the language describer can assert properties of semantic functions and have the assertions checked by an implementation of the type system.

The type system has been implemented for use in connection with the recently developed formalism ASDF. The formalism supports writing language definitions by combining modules that describe single language constructs. This is possible due to the inherent modularity in ASDF. We show how we manage to preserve the modularity and still perform specialised type checks for each module.

1 Introduction

This paper is concerned with type checking of action semantic functions as they occur in the ASDF formalism, so we will start by introducing Action Semantics and ASDF before we return to introducing type checking.

1.1 Action Semantics

Action Semantics (AS) is a hybrid of Denotational Semantics and Operational Semantics. As in a conventional denotational description, inductively defined semantic functions map programs (and declarations, expressions, statements, etc.) compositionally to their denotations, which model their behavior. The difference is that here denotations are *actions* instead of higher-order functions.

An Action Semantic Description (ASD) of a programming language must describe the syntax of the language, semantic functions mapping the language constructs to actions, and semantic entities used in the semantic functions. ASDs of non trivial languages, like Java [3] and SML [11], have already been constructed.

Actions are expressed in Action Notation (AN) [12, 14], a notation resembling English but still strictly formal. AN consists of a *kernel* that is defined operationally; the rest of AN can be reduced to kernel notation. Actions are constructed from yielders, action constants, and action combinators, where yielders consist of data, data operations, and predicates. Yielders are not part of the kernel.

The performance of an action might be seen as an evaluation of a function from data and bindings to data, with side effects like changing storage and sending messages. We shall often refer to the input data/bindings of an action as the *given data/bindings*. The action combinators correspond to different ways of composing functions to obtain different kinds of control and data flow in the evaluation. The evaluation can terminate in three different ways: *Normally* (the performance of the enclosing action continues normally), *abruptly* (the enclosing action is skipped until the exception is handled), or *failing* (corresponding to abandoning the current alternative of a choice and trying alternative actions). AN has actions to represent evaluation of expressions, declarations, abstractions, manipulation of storage, and communication between agents. The yielders can be used to inspect memory locations and compute data and bindings.

To limit this paper, we are not concerned with the actions used to represent communication between agents. Table 1 presents all kernel action combinators and constants, together with a short informal explanation. In the figure A ranges over actions.

Action	**Explanation**
copy	returns the given data
result D	returns data D
give O	applies data operator O to the given data
A_1 then A_2	output from A_1 is input to A_2
A_1 and-then A_2	sequencing, results are concatenated
A_1 and A_2	interleaving, results are concatenated
indivisibly A	A cannot be interleaved with other actions
check O	terminates abruptly if O returns $false$
choose-nat	returns a random non-negative integer
unfolding A	iterates A (in combination with unfold)
unfold	performs action A of smallest enclosing unfolding A
throw	terminates abruptly with the given data as result
A_1 catch A_2	A_2 receives output if A_1 terminates abruptly
A_1 and-catch A_2	abrupt sequencing, results are concatenated
fail	fails
A_1 else A_2	A_2 is the alternative if A_1 fails
copy-bindings	returns current bindings as data
A_1 scope A_2	the scope of bindings produced by A_1 is A_2
recursively A	allows recursive bindings in A
apply	applies the given action to the given data
close	computes the closure of the given action
create	allocates a fresh location
inspect	inspects the contents of the given location
update	updates the given location with the given data

Table 1: Kernel AN

Fig. 1 gives an example of an action. In line 1 a new memory location l_1, containing a random non-negative integer, is allocated. In line 3 the identifier "x" is provided, and the action combinator in line 2 makes sure that line 3 is performed after line 1 and that the output from both evaluations is concatenated into the tuple (x, l_1). Line 4 passes the tuple to the action in line 5 which applies the data operator binding to it and returns the bindings map $\{$ x : l_1 $\}$. The scope of these bindings is line 7 where they are just returned as data.

```
      (((result x)
    and-then
      (choose-nat then create))
  then
    (give binding))
scope
    copy-bindings
```

Figure 1: Example of an action

1.2 ASDF

The recent formalism ASDF [2] was developed by Peter Mosses and the author. The main purpose of ASDF is to support writing action semantic descriptions of single language constructs [7]. These constructs can then be combined into a description of a full language. Since an ASDF module only describes a single language construct, and ASDF has inherent modularity, the modules can easily be reused verbatim in other language descriptions.

An ASDF module consists of *syntax*, *semantics*, and *requires* sections, defining the abstract syntax of the language construct[1], the semantic function mapping the construct to an action, and auxiliary notation used in the semantic function respectively.

The module in Fig. 2 exemplifies the use of the *syntax* and *semantics* sections to define the *local definitions* construct.

In Fig. 3 the requires section is used to declare variables with prefix '`E`' to range over the syntactic sort *Exp*. This declaration is utilised in Fig. 2 where the semantic equation uses variables to range over sub-trees. The requires section in Fig. 5 is used to declare a datatype *Func* with a data constructor *func* and a data selector *action*. This is further explained in Subsection 3.1.

Based on the syntactic sorts used in a module other modules are imported that define the common properties of these sorts, e.g., if the sort *Exp* is used the module *Exp* is imported. The module in Fig. 2 imports *Exp* (Fig. 3).

A complete language description consists of a collection of ASDF modules. Since we only deal with abstract syntax, a complete language description must also contain a mapping from concrete syntax to abstract syntax. For

[1] Concrete syntax is not supported because it impedes reuse of the module, e.g., "`if` *Exp* `then` *Exp* `else` *Exp*" in ML has the same semantics as "*Exp* `?` *Exp* `:` *Exp*" in C, but not the same concrete syntax

```
module Exp/Local

syntax

  Exp ::= local(Dec, Exp)

semantics

  evaluate local(D, E) = furthermore declare D
                         scope evaluate E
```

Figure 2: The *Exp/Local* module

```
module Exp

requires

  E : Exp

semantics

  evaluate: Exp -> using data & giving val
```

Figure 3: The *Exp* module

```
module Dec

requires

  D : Dec

semantics

  declare: Dec -> using data & giving bindings
```

Figure 4: The *Dec* module

this purpose we have found the ASF+SDF formalism [6] helpful.

```
module Data/Func

requires

  Func ::= func(action: using val & giving val & raising val)
```

Figure 5: The *Data/Func* module

1.3 Type checking

Type checking in connection with AS can be done at two levels: Either semantic functions in an ASD are type checked to reveal mistakes made by the language describer, or the actions resulting from applying the semantic functions to a program are type checked to reveal errors in the program and to support code generation (if a type has been inferred for all subactions). The topic of this chapter is the former.

Type checking a semantic function is done one module at a time by type checking the semantic equations in a module defining the semantic function's behaviour on a single construct. By type checking semantic equations we mean checking that the equation conforms to the signature of the semantic function it defines. This of course requires information about all the user defined semantic functions, types, data, and data operators used in a semantic equation, and it requires that information can be collected from the module containing the equation and the modules it imports. If the action in a semantic equation contains type errors or its type is not a subtype of the expected type (according to the semantic function signature), we can report an error. We shall say that an action has a type error if one of its subactions terminates abruptly because it is given a type of data it did not expect. An example of an action containing a type error is the action '$\mathsf{result\ 5\ then\ close}$'. This action is flawed because close expects an action, but receives an integer. If $\mathsf{execute}$ has the signature '$\mathsf{execute} : \mathit{Stm} \rightarrow \mathsf{Action\ \&\ using\ data\ \&\ giving\ ()}$', the semantic equation "$\mathsf{execute\ new}(E) = \mathsf{evaluate}\ E\ \mathsf{then\ create}$" will also result in a type error because the signature does not allow that actions resulting from $\mathsf{execute}$ give memory cells.

Type checking of semantic equations is obstructed by the fact that the action on the right hand side appears out of the context it will appear in when the semantic equation is used to map a complete program. A conservative type checker would reject many semantic equations because of the lack of information about the context. It is worth considering whether we can use quantified types or principal typings [15] to solve the problem with the missing context. We could use quantified types where we quantify over the tokens bound in the context, or we could use principal typings to define type judgements that describe the context, but the main problem is that type checking is done before token values are known (they will not be known until the semantic functions are applied to concrete programs), so we can never instantiate the quantified types or check the type judgements. Therefore quantified types or principal typings would not help us. We have chosen to develop a soft type checker that approves many actions but still warns the user against the most obvious mistakes.

The purpose of the type checker is to type check semantic functions in ASDF modules. Because of the modularity of ASDF descriptions we also want the type checker to be modular as explained in Section 5.

An implementation of the type checker, integrated into the Action Environment [2], has been used to type check an ASDF description of the core of ML [11].

1.4 Overview of the paper

In Section 2 we present related work. In Section 3 we present the types, operators on types, and the type rules that make up our type system. An example of type checking can be found in Section 4. Modularity is discussed in Section 5. In Section 6 we briefly describe how we have implemented the type system. Section 8 concludes.

2 Related work

Type checking (or type inference) of AN has been a research area since the beginning of the 1990's where Even and Schmidt [9] showed how to infer types for actions using unification on record types. Their work has been further developed by Brown [4], Lee [13], and Iversen [10]. Common to all these systems is that the goal is to infer a type for a self-contained action for use in code generation. This differs from what we will present in this chapter in that we want to type check semantic functions where the embodied action describes a small part of a full program, and the main goal is to give the language developer useful feedback about his description and let him test assertions about semantic functions.

Doh and Schmidt [8] describe a method for extracting typing laws from semantic functions. This is not type checking of the semantic functions, but a way to compute type rules for the described language from the semantic functions.

In [16] Ørbæk describes a soft type inference algorithm for semantic functions. The algorithm is not dependent on the user giving any kind of type annotations, like signatures, to the semantic functions; instead it infers a type by looking at all the semantic equations in the language description. This differs from our approach because we want to type check the semantic equations that describe a single language construct without looking at the whole language description.

3 Type system

Our type system consists of a set of types $\mathcal{T}$ ordered by a subtype relation and a set of type rules that can be used to derive a proof that an action has a certain type. We will present both in the following two sub-sections. Throughout the rest of this chapter we will use τ as a variable that ranges over types.

3.1 Types

We shall view types as sets of values. Our type system has three different kinds of types: the built-in AN types, the action types, and the user defined types. The built-in AN types are listed in Fig. 6.

$$\begin{array}{rcl} \mathit{Type} & ::= & \mathsf{data} \mid \mathsf{datum} \mid \emptyset \mid \mathsf{integer} \mid \mathsf{boolean} \mid \mathsf{token} \mid \mathsf{bindable} \\ & & \mathsf{bindings} \mid \mathsf{storable} \mid \mathsf{cell} \mid \mathit{ActionType} \mid \mathit{Type} \times \ldots \times \mathit{Type} \end{array}$$

Figure 6: AN built-in types

The type data contains all values, and all types are subtypes of data. All values except tuples of data is included in the type datum. The type $\emptyset$ does not contain any values. Notice that action types (*ActionType*) are also included in the built-in types; this is necessary because actions can be used as data in AN. The product type is the type of tuples of data, and the symbol () denotes the product type of length 0, the type of the empty tuple (like `unit` in Standard ML).

$$\begin{array}{rcl} \mathit{ActionType} & ::= & \mathsf{Action} \mid \mathsf{using}\ \mathit{Type} \mid \mathsf{giving}\ \mathit{Type} \mid \mathsf{raising}\ \mathit{Type} \mid \\ & & \mathsf{infallible} \mid \mathsf{closed} \mid \mathsf{terminates} \mid \mathsf{uncreative} \mid \\ & & \mathsf{ineffective} \mid \mathsf{stable} \mid \mathit{ActionType}\ \&\ \mathit{ActionType} \end{array}$$

Figure 7: Action types

Action types are listed in Fig. 7. We use the symbol & to denote the intersection of two action types. The type Action is the supertype of all action types and says nothing about the action, except that it is an action. The three types parameterised with a type, '$\mathsf{using}\ \tau$', '$\mathsf{giving}\ \tau$', and '$\mathsf{raising}\ \tau$', are the types for actions that can be given data of some type, actions that produce data of some type when they terminate normally, or actions that produce data of some type in case of abrupt termination, respectively.

An action type which does not contain '$\mathsf{using}\ \tau$', '$\mathsf{giving}\ \tau$', or '$\mathsf{raising}\ \tau$' is equal to the same action type with '$\mathsf{using}\ \emptyset$', '$\mathsf{giving\ data}$', or '$\mathsf{raising\ data}$' respectively added (this means that '$\mathsf{Action} \equiv \mathsf{using}\ \emptyset\ \&\ \mathsf{giving\ data}\ \&\ \mathsf{raising\ data}$'). This is also illustrated in the equivalence in Fig. 8. This equivalence is a consequence of '$\mathsf{using}\ \tau$' being contravariant in its type argument and '$\mathsf{giving}\ \tau$' and '$\mathsf{raising}\ \tau$' being covariant, as shown in the subtype relations listed in Fig. 9. Throughout the rest of this chapter we shall use α as a variable to range over all action types and γ to range over atomic action types (all action types listed in Fig. 7 except '$\mathit{ActionType}\ \&\ \mathit{ActionType}$').

The type '$\mathsf{using\ data}$' contains only the actions which accept all types of input. Many of the actions with this type ignore their input, like '$\mathsf{result}\ D$'. The types '$\mathsf{giving}\ \emptyset$' and '$\mathsf{raising}\ \emptyset$' contain the actions that cannot terminate normally or abruptly, respectively.

The names of the rest of the types should indicate what their intended meaning is. To illustrate their use, the action type '$\mathsf{giving\ token} \times \mathsf{bindings}\ \&\ \mathsf{infallible}\ \&\ \mathsf{stable}$' describes the actions which produce a pair consisting of a token and bindings, and do not fail or inspect memory. The action type '$\mathsf{using\ storable}\ \&\ \mathsf{closed}\ \&\ \mathsf{ineffective}\ \&\ \mathsf{uncreative}$' describes actions which can be given a storable, are closed with respect to bindings, do not update storage, and do not allocate new memory locations.

Some of the action types are "negative" in the way that they describe behaviour an action may *not* have: it may *not* fail ($\mathsf{infallible}$), it may *not* create new memory cells ($\mathsf{uncreative}$), it may *not* update memory ($\mathsf{ineffective}$), or it may *not* inspect memory cells (stable). The reason we have chosen "negative" types in these cases is that it is difficult (often impossible) to determine if an action fails or manipulates storage. It is difficult because we cannot with static analysis determine which parts of an action are evaluated. On the other hand we can easily point out a large set of actions that, for instance, do not create memory cells (the actions that do not contain the action create). This also means that if an action type does not contain, for instance, $\mathsf{infallible}$, it describes all actions that might fail or not fail.

Fig. 8 presents an equivalence on action types. The five rules say that the order of the atomic action types is not important, the action types '$\mathsf{using}\ \emptyset$', '$\mathsf{giving\ data}$', and '$\mathsf{raising\ data}$' can be introduced, and if there is a subtype relation between two atomic action types the "highest" type can be removed (this also means that if an atomic action type occurs twice in an action type one of the occurrences can be removed). When a type operator or a type rule mentions an action type, we shall assume that the equivalence has been applied to the action type such that the type operator or the type rule can be applied.

$$\gamma_1 \mathbin{\&} \ldots \mathbin{\&} \gamma_{i-1} \mathbin{\&} \gamma_i \mathbin{\&} \ldots \mathbin{\&} \gamma_n \equiv \gamma_1 \mathbin{\&} \ldots \mathbin{\&} \gamma_i \mathbin{\&} \gamma_{i-1} \mathbin{\&} \ldots \mathbin{\&} \gamma_n$$

$$\alpha \equiv \mathsf{using}\ \emptyset \mathbin{\&} \alpha$$

$$\alpha \equiv \mathsf{giving\ data} \mathbin{\&} \alpha$$

$$\alpha \equiv \mathsf{raising\ data} \mathbin{\&} \alpha$$

$$\gamma_1 \mathbin{\&} \gamma_2 \mathbin{\&} \ldots \mathbin{\&} \gamma_n \equiv \gamma_2 \mathbin{\&} \ldots \mathbin{\&} \gamma_n \quad \textit{when} \quad \gamma_2 \leq \gamma_1$$

Figure 8: Equivalence on action types

The readers familiar with the previous work on inferring types for actions [4, 9, 10, 13] might have noticed that our types cannot describe the bindings used by an action. In previous work the bindings used by an action were also inferred using record types. This allowed a stronger type inference because the type of the output from yielders, like '$\mathsf{bound\text{-}to\ the\ token\ x}$', was more specific than just bindable, and the type inference algorithm was able to check that the token was actually bound in the current bindings. Due to the fact that token values are seldom known in a semantic function before the function is applied, the type system cannot deal with bindings on a more detailed level than the atomic type bindings. To illustrate this, a semantic function containing the action 'give the bindable bound-to I' (where I is an ASDF variable ranging over tokens) will always type check in our system because the value of I is not known until the semantic function is applied, so we cannot check that the instantiation of I is bound in the current bindings.

In an ASDF module the user can provide type information. A production, like '*Bindable* ::= *Integer*' defines the type integer to be a subtype of bindable[2]. A more advanced production, like '*Func* ::= func(action : using val & giving val & raising val)' (see Fig. 5), defines the data constructor func to be a data operator which takes an action of type 'using val & giving val & raising val' and gives data of type func. The production also defines the data selector action to be a data operator which takes a func and gives an action of the before mentioned type. Finally ASDF modules can also contain signatures of semantic functions, like 'evaluate : *Exp* $\rightarrow$ Action & using data & giving val' (see Fig. 3).

As mentioned before, the set of types $\mathcal{T}$ is ordered, and the ordering $\leq$ is defined in Fig. 9.

3.2 Type rules

Type rules can be used to construct a proof that an action has a certain type, and from type rules type inference rules can be constructed. For an algorithm that checks that an action has a certain type, see Section 6.

In Figs. 11 and 12 we see examples of type rules for the actions used to describe normal flow of data and control in programming languages. The rules are conditional as illustrated in rule 2 where the premises state the types of the two subactions A_1 and A_2. In all type rules for action combinators the premises will state what the types of the subactions are. Rule 2 also has other conditions which state that the type of the data produced by A_1 should not be $\emptyset$ (recall that 'giving $\emptyset$' means that the action does not terminate normally and then the right subaction would never be executed, which we consider an

[2]The convention in ASDF is to use words starting with capital letters for naming syntactic sorts whereas AN uses small letters in types

$$\tau \leq \tau$$

$$\tau_1 \leq \tau_2 \ \wedge \ \tau_2 \leq \tau_3 \ \Rightarrow \ \tau_1 \leq \tau_3$$

$$\emptyset \leq \tau$$

$$\tau \leq \mathsf{datum} \quad \mathit{when} \quad \tau \neq () \wedge \forall n \geq 2, \tau_i.\ \tau \neq \tau_1 \times \ldots \times \tau_n$$

$$\tau \leq \mathsf{data}$$

$$\alpha \leq \mathsf{Action}$$

$$\alpha \leq \gamma_1 \ \& \ \ldots \ \& \ \gamma_n \quad \mathit{when} \quad \forall i \in 1..n.\ \alpha \leq \gamma_i$$

$$\gamma_1 \ \& \ \ldots \ \& \ \gamma_n \leq \gamma \quad \mathit{when} \quad \exists i \in 1..n.\ \gamma_i \leq \gamma$$

$$\mathsf{using}\ \tau_1 \leq \mathsf{using}\ \tau_2 \quad \mathit{when} \quad \tau_2 \leq \tau_1$$

$$\mathsf{giving}\ \tau_1 \leq \mathsf{giving}\ \tau_2 \quad \mathit{when} \quad \tau_1 \leq \tau_2$$

$$\mathsf{raising}\ \tau_1 \leq \mathsf{raising}\ \tau_2 \quad \mathit{when} \quad \tau_1 \leq \tau_2$$

$$\tau_1 \times \ldots \times \tau_n \leq \tau_1' \times \ldots \times \tau_n' \quad \mathit{when} \quad \forall i \in 1..n.\ \tau_i \leq \tau_i'$$

+ user defined relations in ASDF modules

Figure 9: Subtype relation

$$\mathsf{simple} = \ \mathsf{infallible} \ \& \ \mathsf{closed} \ \& \ \mathsf{terminates} \ \& \ \mathsf{uncreative} \ \& \ \mathsf{ineffective} \ \& \ \mathsf{stable} \tag{1}$$

Figure 10: Definition of simple

error). The condition '$\tau_1' \leq \tau_2$' states that the type of data produced by

$$\frac{\begin{array}{c}\alpha_u \vdash A_1 : \mathsf{using}\ \tau_1\ \&\ \mathsf{giving}\ \tau_1'\ \&\ \mathsf{raising}\ \tau_1^r\ \&\ \alpha_1 \\ \alpha_u \vdash A_2 : \mathsf{using}\ \tau_2\ \&\ \mathsf{giving}\ \tau_2'\ \&\ \mathsf{raising}\ \tau_2^r\ \&\ \alpha_2 \\ \tau_1' \leq \tau_2,\ \ \tau_1' \neq \emptyset\end{array}}{\begin{array}{rl}\alpha_u \vdash A_1\ \mathsf{then}\ A_2 : & \mathsf{using}\ \tau_1\ \&\ \mathsf{giving}\ \tau_2'\ \& \\ & \mathsf{raising}\ (\tau_1^r \cup \tau_2^r)\ \&\ (\alpha_1 \cup_{ac} \alpha_2)\end{array}} \tag{2}$$

$$\frac{\begin{array}{c}\alpha_u \vdash A_1 : \mathsf{using}\ \tau_1\ \&\ \mathsf{giving}\ \tau_1'\ \&\ \mathsf{raising}\ \tau_1^r\ \&\ \alpha_1 \\ \alpha_u \vdash A_2 : \mathsf{using}\ \tau_2\ \&\ \mathsf{giving}\ \tau_2'\ \&\ \mathsf{raising}\ \tau_2^r\ \&\ \alpha_2\end{array}}{\begin{array}{rl}\alpha_u \vdash A_1\ \mathsf{and}\ A_2 : & \mathsf{using}\ (\tau_1 \cap \tau_2)\ \&\ \mathsf{giving}\ (\tau_1' \oplus \tau_2')\ \& \\ & \mathsf{raising}\ (\tau_1^r \cup \tau_2^r)\ \&\ (\alpha_1 \cup_{ac} \alpha_2)\end{array}} \tag{3}$$

$$\frac{\begin{array}{c}\alpha_u \vdash A_1 : \mathsf{using}\ \tau_1\ \&\ \mathsf{giving}\ \tau_1'\ \&\ \mathsf{raising}\ \tau_1^r\ \&\ \alpha_1 \\ \alpha_u \vdash A_2 : \mathsf{using}\ \tau_2\ \&\ \mathsf{giving}\ \tau_2'\ \&\ \mathsf{raising}\ \tau_2^r\ \&\ \alpha_2\end{array}}{\begin{array}{rl}\alpha_u \vdash A_1\ \mathsf{and\text{-}then}\ A_2 : & \mathsf{using}\ (\tau_1 \cap \tau_2)\ \&\ \mathsf{giving}\ (\tau_1' \oplus \tau_2')\ \& \\ & \mathsf{raising}\ (\tau_1^r \cup \tau_2^r)\ \&\ (\alpha_1 \cup_{ac} \alpha_2)\end{array}} \tag{4}$$

$$\alpha_u \vdash \mathsf{copy} : \mathsf{using}\ \tau\ \&\ \mathsf{giving}\ \tau\ \&\ \mathsf{raising}\ \emptyset\ \&\ \mathsf{simple} \tag{5}$$

Figure 11: Type rules for normal flow of data and control AN

A_1 should be a subtype of the type of data that can be given to A_2. If the premises hold, we can derive the type of 'A_1 then A_2' using the types from the premises and appropriate type operators to combine them. The $\cup$ ($\cap$) operator computes the union (intersection) of two types, and the $\cup_{ac}$ operator takes two action types and returns the intersection of the atomic action types occurring in both the action types. In other words '$\alpha_1 \cup_{ac} \alpha_2$' is the lowest action type bigger than α_1 and α_2, i.e., a union on action types rounded up to the nearest action type (a least upper bound).

The rule for the action combinator and (rule 3) introduces the type operator $\oplus$ which concatenates two types into a product type. A formal definition of all the type operators can be found in Fig. 13.

The rules for give (rule 7 and 8) use the constant simple which is an abbreviation for an action type. The expansion can be found in rule 1 in Fig. 10. The type of give O depends on the signature of the data operator O where the question mark indicates that it is a partial operator. The action 'check O' (rule 9) also contains a data operator, but the rule does not depend

$$\frac{D : \tau}{\alpha_u \vdash \textsf{result } D : \textsf{using data} \mathrel{\&} \textsf{giving } \tau \mathrel{\&} \textsf{raising } \emptyset \mathrel{\&} \textsf{simple}} \tag{6}$$

$$\frac{O : \tau \to ?\ \tau'}{\alpha_u \vdash \textsf{give } O : \textsf{using } \tau \mathrel{\&} \textsf{giving } \tau' \mathrel{\&} \textsf{raising } () \mathrel{\&} \textsf{simple}} \tag{7}$$

$$\frac{O : \tau \to \tau'}{\alpha_u \vdash \textsf{give } O : \textsf{using } \tau \mathrel{\&} \textsf{giving } \tau' \mathrel{\&} \textsf{raising } \emptyset \mathrel{\&} \textsf{simple}} \tag{8}$$

$$\frac{O : \tau \to \textsf{boolean}}{\alpha_u \vdash \textsf{check } O : \textsf{using } \tau \mathrel{\&} \textsf{giving } \tau \mathrel{\&} \textsf{raising } () \mathrel{\&} \textsf{simple}} \tag{9}$$

$$\frac{\alpha_u \vdash A : \alpha}{\alpha_u \vdash \textsf{indivisibly } A : \alpha} \tag{10}$$

$$\alpha_u \vdash \textsf{choose-nat} : \textsf{using data} \mathrel{\&} \textsf{giving integer} \mathrel{\&} \textsf{raising } \emptyset \mathrel{\&} \textsf{simple} \tag{11}$$

$$\frac{\alpha' \vdash A : \alpha'}{\alpha_u \vdash \textsf{unfolding } A : \alpha'} \tag{12}$$

$$\frac{\textsf{terminates} \notin \alpha_u}{\alpha_u \vdash \textsf{unfold} : \alpha_u} \tag{13}$$

Figure 12: Type rules for normal flow of data and control AN (continued)

on whether the operator is partial since '$\textsf{check } O$' can still terminate abruptly when the data operator is not partial. The rule insists that the result type of the operator is $\textsf{boolean}$.

Each rule has an action type on the left hand side of the turnstile, and most of the rules just propagate it to the premises. The action type is used in connection with the two actions related to unfolding (rule 12 and 13). The type of '$\textsf{unfolding } A$' is the same as the type of A, and the type of $\textsf{unfold}$ is the

Type concatenation

$$\begin{aligned}\tau_1 \oplus \tau_2 &= \tau_1 \times \tau_2\\ \emptyset \oplus \tau &= \emptyset\\ \tau \oplus \emptyset &= \emptyset\end{aligned}$$

Action type combinator

$$\begin{array}{rcll}(\gamma \ \& \ \alpha_1) \cup_{ac} (\gamma \ \& \ \alpha_2) &=& \gamma \ \& \ (\alpha_1 \cup_{ac} \alpha_2) & \\ (\gamma \ \& \ \alpha_1) \cup_{ac} \alpha_2 &=& (\alpha_1 \cup_{ac} \alpha_2) & \mathit{when} \quad \gamma \notin \alpha_2\\ \gamma \cup_{ac} (\gamma \ \& \ \alpha) &=& \gamma & \\ \gamma \cup_{ac} \alpha &=& \mathsf{Action} & \mathit{when} \quad \gamma \notin \alpha\end{array}$$

Action type subtraction

$$\begin{array}{rcll}(\gamma \ \& \ \alpha)\backslash\gamma &=& \alpha\backslash\gamma & \\ (\gamma_1 \ \& \ \alpha)\backslash\gamma_2 &=& \gamma_1 \ \& \ (\alpha\backslash\gamma_2) & \mathit{when} \quad \gamma_1 \neq \gamma_2\\ \gamma\backslash\gamma &=& \mathsf{Action} & \\ \gamma_1\backslash\gamma_2 &=& \gamma_1 & \mathit{when} \quad \gamma_1 \neq \gamma_2\end{array}$$

Action type projections

$$\mathit{has}(\gamma, \alpha) = \begin{cases}\gamma & \mathit{when} \quad \gamma \in \alpha\\ \mathsf{Action} & \mathit{when} \quad \gamma \notin \alpha\end{cases}$$

Figure 13: Type operators

same as the type of the enclosing unfolding action. When using the rule for unfolding, the type of A must be guessed and then passed on to the premise that derives the type for A. The rule for unfold just states that unfolds's type is the type left to the turnstile and that this type cannot contain terminates.

The rules in Fig. 14 concern the actions used to describe exceptional and alternative control flow (like raising exceptions and conditional expressions). Comparing with Fig. 11 we see that there are many similarities (compare throw with copy, then with catch, and and with and-catch). The main difference is that some actions terminate abruptly instead of normally.

Intersection between action types is very common in our type system, but as illustrated in the rule for fail (rule 17), there is also a subtraction operator \. The action fail does of course fail, and therefore we must remove the type infallible from its type.

In rule 18 we introduce the type operator *has*. The domain of *has* is an

$$\alpha_u \vdash \mathsf{throw} : \mathsf{using}\ \tau\ \&\ \mathsf{giving}\ \emptyset\ \&\ \mathsf{raising}\ \tau\ \&\ \mathsf{simple} \tag{14}$$

$$\frac{\begin{array}{c}\alpha_u \vdash A_1 : \mathsf{using}\ \tau_1\ \&\ \mathsf{giving}\ \tau_1'\ \&\ \mathsf{raising}\ \tau_1^r\ \&\ \alpha_1 \\ \alpha_u \vdash A_2 : \mathsf{using}\ \tau_2\ \&\ \mathsf{giving}\ \tau_2'\ \&\ \mathsf{raising}\ \tau_2^r\ \&\ \alpha_2 \\ \tau_1^r \leq \tau_2,\ \tau_1^r \neq \emptyset\end{array}}{\begin{array}{l}\alpha_u \vdash A_1\ \mathsf{catch}\ A_2 : \ \mathsf{using}\ \tau_1\ \&\ \mathsf{giving}\ (\tau_1' \cup \tau_2')\ \& \\ \qquad\qquad\qquad\quad \mathsf{raising}\ \tau_2^r\ \&\ (\alpha_1 \cup_{ac} \alpha_2)\end{array}} \tag{15}$$

$$\frac{\begin{array}{c}\alpha_u \vdash A_1 : \mathsf{using}\ \tau_1\ \&\ \mathsf{giving}\ \tau_1'\ \&\ \mathsf{raising}\ \tau_1^r\ \&\ \alpha_1 \\ \alpha_u \vdash A_2 : \mathsf{using}\ \tau_2\ \&\ \mathsf{giving}\ \tau_2'\ \&\ \mathsf{raising}\ \tau_2^r\ \&\ \alpha_2\end{array}}{\begin{array}{l}\alpha_u \vdash A_1\ \mathsf{and\text{-}catch}\ A_2 : \mathsf{using}\ (\tau_1 \cap \tau_2)\ \&\ \mathsf{giving}\ (\tau_1' \cup \tau_2')\ \& \\ \qquad\qquad\qquad\qquad\qquad \mathsf{raising}\ (\tau_1^r \oplus \tau_1^r)\ \&\ (\alpha_1 \cup_{ac} \alpha_2)\end{array}} \tag{16}$$

$$\alpha_u \vdash \mathsf{fail} : \mathsf{using\ data}\ \&\ \mathsf{giving}\ \emptyset\ \&\ \mathsf{raising}\ \emptyset\ \&\ \mathsf{simple} \setminus \mathsf{infallible} \tag{17}$$

$$\frac{\begin{array}{c}\alpha_u \vdash A_1 : \mathsf{using}\ \tau_1\ \&\ \mathsf{giving}\ \tau_1'\ \&\ \mathsf{raising}\ \tau_1^r\ \&\ \alpha_1 \\ \alpha_u \vdash A_2 : \mathsf{using}\ \tau_2\ \&\ \mathsf{giving}\ \tau_2'\ \&\ \mathsf{raising}\ \tau_2^r\ \&\ \alpha_2 \\ \mathsf{infallible} \notin \alpha_1\end{array}}{\begin{array}{l}\alpha_u \vdash A_1\ \mathsf{else}\ A_2 : \mathsf{using}\ (\tau_1 \cap \tau_2)\ \&\ \mathsf{giving}\ (\tau_1' \cup \tau_2')\ \& \\ \qquad\qquad\qquad\qquad \mathsf{raising}\ (\tau_1^r \cup \tau_2^r)\ \&\ (\alpha_1 \cup_{ac} \alpha_2)\ \&\ has(\mathsf{infallible}, \alpha_2)\end{array}} \tag{18}$$

Figure 14: Type rules for abrupt and alternative control flow AN

atomic action type and an action type. The operator returns the first type if the second action type contains the first, otherwise the result is Action. Using this operator ensures that the type of the whole action contains infallible if the right subaction cannot fail.

In Fig. 15 the type rules for actions describing declarations are shown. The atomic type bindings is used in all three rules to describe that an action produces a mapping from token's to data. As discussed in Subsection 3.1, action types does not describe the bindings used by an action, but the type closed indicates that an action does not use the current bindings.

$$\frac{\alpha_u \vdash A : \mathsf{giving\ bindings}\ \&\ \alpha'}{\alpha_u \vdash \mathsf{recursively}\ A : \mathsf{giving\ bindings}\ \&\ \alpha'} \tag{19}$$

$$\alpha_u \vdash \mathsf{copy\text{-}bindings} : \begin{array}{l}\mathsf{using\ data}\ \&\ \mathsf{giving\ bindings}\ \&\\ \mathsf{raising}\ \emptyset\ \&\ \mathsf{simple} \setminus \mathsf{closed}\end{array} \tag{20}$$

$$\frac{\begin{array}{c}\alpha_u \vdash A_1 : \mathsf{using}\ \tau_1\ \&\ \mathsf{giving\ bindings}\ \&\ \mathsf{raising}\ \tau_1^r\ \&\ \alpha_1\\ \alpha_u \vdash A_2 : \mathsf{using}\ \tau_2\ \&\ \mathsf{giving}\ \tau_2'\ \&\ \mathsf{raising}\ \tau_2^r\ \&\ \alpha_2\end{array}}{\begin{array}{l}\alpha_u \vdash A_1\ \mathsf{scope}\ A_2 : \mathsf{using}\ (\tau_1 \cap \tau_2)\ \&\ \mathsf{giving}\ \tau_2'\ \&\\ \qquad\qquad \mathsf{raising}\ (\tau_1^r \cup \tau_2^r)\ \&\ (\alpha_1 \cup_{ac} \alpha_2)\ \&\ has(\mathsf{closed}, \alpha_1)\end{array}} \tag{21}$$

Figure 15: Type rules for declarative AN

$$\frac{\tau_2 \leq \tau_1}{\alpha_u \vdash \mathsf{apply} : \mathsf{using}\ ((\mathsf{using}\ \tau_1\ \&\ \alpha') \times \tau_2)\ \&\ \alpha' \setminus \mathsf{terminates}} \tag{22}$$

$$\alpha_u \vdash \mathsf{close} : \mathsf{using}\ \alpha'\ \&\ \mathsf{giving}\ (\alpha'\ \&\ \mathsf{closed})\ \ \&\ \mathsf{raising}\ \emptyset\ \&\ \mathsf{simple} \tag{23}$$

Figure 16: Type rules for reflective AN

Actions can handle other actions as data; this necessitates the inclusion of action types in the set of ordinary types so it becomes a higher-order type system. In Fig. 16 this is illustrated. The actions there expect an action as input and either execute it with some arguments and return the result (rule 22), or they return a moderated action (rule 23). We cannot guarantee that the action apply terminates because it might recur forever, and therefore terminates must be removed from α'. Notice also that the use of the variable α' expresses how the type of apply depends on the type of the action given to apply.

The three rules in Fig. 17 shows the use of the action types uncreative, ineffective, and stable. Besides defining the type of data used, produced, and raised by the actions the rules also illustrate how the three action types are

$$\alpha_u \vdash \mathsf{create} : \mathsf{using\ storable}\ \&\ \mathsf{giving\ cell}\ \&\ \mathsf{raising}\ \emptyset\ \&\ \mathsf{simple} \setminus \mathsf{uncreative} \tag{24}$$

$$\alpha_u \vdash \mathsf{update} : \mathsf{using}\ (\mathsf{cell} \times \mathsf{storable})\ \&\ \mathsf{giving}\ ()\ \&\ \mathsf{raising}\ ()\ \&\ \mathsf{simple} \setminus \mathsf{ineffective} \tag{25}$$

$$\alpha_u \vdash \mathsf{inspect} : \mathsf{using\ cell}\ \&\ \mathsf{giving\ storable}\ \&\ \mathsf{raising}\ ()\ \&\ \mathsf{simple} \setminus \mathsf{stable} \tag{26}$$

Figure 17: Type rules for imperative AN

closely connected to these three actions, i.e., the type of an action contains uncreative, ineffective, or stable if, and only if, it does not contain the actions create, update, or inspect, respectively.

The actions found in semantic functions can contain applications of other semantic functions as subactions (as illustrated in Fig. 2). The type rule for these applications is shown in rule 27. The rule states that if the function f has a signature $\sigma \to \alpha$, then the result of applying f to a construct S of syntactic sort σ has type α.

$$\frac{\begin{array}{c}\alpha_u \vdash f : \Sigma \to \alpha \\ S : \Sigma\end{array}}{\alpha_u \vdash f\ S : \alpha} \tag{27}$$

Figure 18: Type rule for semantic function

The subsumption rule (rule 28 in Fig. 19) says that if an action A has a type α and α' is a supertype of α, then A also has the type α'.

AN contains only few built-in data operators and expects the user to provide the necessary definitions of data and data operators. We shall not spend many lines on data notation here, but it is relevant to know about the built-in partial data operator '$\mathsf{the}\ \tau$' which performs type projections. Given

$$\frac{\begin{array}{c}\alpha_u \vdash A:\ \alpha \\ \alpha \leq \alpha'\end{array}}{\alpha_u \vdash A:\ \alpha'} \tag{28}$$

Figure 19: Subsumption rule

data of type τ it returns the given data, otherwise it is undefined. In our type system we have to settle with the type of data given to 'the τ' not being disjoint with τ because we often cannot determine types that are specific enough. An example of this is the action 'inspect then give the integer' where inspect produces a storable which is not necessarily an integer (but an integer can be a storable). This liberal typing of 'the τ' turns our type checker into a soft type checker, because we cannot always guarantee that the action will not err when it tries to perform 'the τ'.

4 An example

To illustrate the use of the type system we will try to type check the ASDF module *Exp/Local* (see Fig. 2). The module describes declarations local to an expression. To maintain simplicity we have omitted 'raising τ' and the action before the turnstile from the rules in this example. Before type checking can start, the type information defined by the user must be collected. The type information relevant for the module *Exp/Local* can be found in the modules *Exp* (see Fig. 3) and *Dec* (see Fig. 4).

It is also necessary to rewrite the action in the semantic equation to the corresponding kernel action. This is necessary because we only have type rules for kernel actions. The action

(furthermore (declare *D*)) scope (evaluate *E*)

corresponds to the kernel action

(1) ((copy-bindings and (declare *D*))
(2) then
(3) (give overriding))
(4) scope
(5) (evaluate *E*)

Starting from the top of the parse tree representing the kernel action we apply rule 21, the type rule for the action combinator scope. The rule requires a type for the two subactions, so we use rule 2 to derive a type for the left subaction (lines 1-3), and again we must infer a type for the two subactions (lines 1 and 3). To infer a type for the action in line 1 we use rule 20:

$$\vdash \mathsf{copy\text{-}bindings} : \ \mathsf{using\ data} \ \& \ \mathsf{giving\ bindings} \ \& \ \mathsf{simple} \setminus \mathsf{closed} \tag{29}$$

and rule 27

$$\frac{\begin{array}{l}\vdash \mathsf{declare} : \mathit{Dec} \rightarrow \mathsf{using\ data} \ \& \ \mathsf{giving\ bindings} \\ D : \mathit{Dec}\end{array}}{\vdash \mathsf{declare}\ D : \mathsf{using\ data} \ \& \ \mathsf{giving\ bindings}} \tag{30}$$

(where we use the signature from Fig. 4 to satisfy the premises) and finally rule 3 (29 together with 30 provide a proof for the premises).

$$\frac{\begin{array}{l}\vdash \mathsf{copy\text{-}bindings} : \ \mathsf{using\ data} \ \& \ \mathsf{giving\ bindings} \ \& \ \mathsf{simple} \setminus \mathsf{closed} \\ \vdash \mathsf{declare}\ D : \mathsf{using\ data} \ \& \ \mathsf{giving\ bindings}\end{array}}{\begin{array}{l}\vdash \mathsf{copy\text{-}bindings\ and\ (declare}\ D) : \\ \qquad \mathsf{using\ data} \ \& \ \mathsf{giving\ (bindings, bindings)}\end{array}} \tag{31}$$

The data operator $\mathsf{overriding}$ has signature

$$\mathsf{overriding} : \mathsf{bindings} \times \mathsf{bindings} \rightarrow \mathsf{bindings} \tag{32}$$

and using rule 8 we get

$$\frac{\mathsf{overriding} : \mathsf{bindings} \times \mathsf{bindings} \rightarrow \mathsf{bindings}}{\begin{array}{l}\vdash \mathsf{give\ overriding} : \\ \qquad \mathsf{using\ (bindings} \times \mathsf{bindings)} \ \& \ \mathsf{giving\ bindings} \ \& \ \mathsf{simple}\end{array}} \tag{33}$$

Now combining 31, 33, and rule 2 we get:

$$\frac{\begin{array}{l}\vdash A_1 : \mathsf{using\ data} \ \& \ \mathsf{giving\ (bindings, bindings)} \\ \vdash \mathsf{give\ overriding} : \ \mathsf{using\ (bindings,\ bindings)} \ \& \\ \qquad\qquad\qquad\qquad \mathsf{giving\ bindings} \ \& \ \mathsf{simple} \\ \mathsf{bindings} \times \mathsf{bindings} \leq \mathsf{bindings} \times \mathsf{bindings}, \\ \mathsf{bindings} \times \mathsf{bindings} \neq \emptyset\end{array}}{\vdash A_1\ \mathsf{then\ (give\ overriding)} : \ \mathsf{using\ data} \ \& \ \mathsf{giving\ bindings}} \tag{34}$$

(where A_1 is '$\textsf{copy-bindings and (declare } D\textsf{)}$'). The application of the semantic function $\textsf{evaluate}$ in line 5 can be typed using rule 27:

$$\frac{\begin{array}{l}\vdash \textsf{evaluate} : \mathit{Exp} \rightarrow \textsf{using data \& giving val} \\ E : \mathit{Exp}\end{array}}{\vdash \textsf{evaluate } E : \textsf{using data \& giving val}} \tag{35}$$

Finally we can use 34, 35, and rule 21 to derive a type for the whole action (lines 1-5).

$$\frac{\begin{array}{l}\vdash A_{1-3} : \textsf{using data \& giving bindings} \\ \vdash \textsf{evaluate } E : \textsf{using data \& giving val}\end{array}}{\vdash A_{1-3} \textsf{ scope (evaluate } E\textsf{)} : \textsf{using data \& giving val}} \tag{36}$$

where A_{1-3} is the part of the whole action that spans lines 1-3. We now have a type for the action from the right-hand side of the semantic equation in the module *Exp/Local*, and we can conclude the type check by checking that the inferred type is a subtype of the action type in the signature for evaluate. Since they are both 'using data & giving val', we conclude that the semantic equation type checks.

5 Constructive type checking

A constructive ASD of a programming language written in ASDF is extensible and reusable. This is advantageous because it allows incremental development of descriptions, e.g., we can start by describing the core of a language and then incrementally add more features to the language by adding more modules. Furthermore the modules can be reused by reference in other language descriptions.

This section deals with the problem that we might want different signatures for the same semantic function depending on which properties we want, for instance, expressions to have in our description. The problem is complicated further because we want the ASDF modules to preserve their reusability.

In a typical description of a language we have a module *Exp* containing all the features common to expressions as illustrated in Fig. 3. This module is then imported (automatically) from all modules describing expressions. In *Exp* we put the signature of the semantic function evaluate which maps expressions to actions. The signature requires that the action resulting from applying evaluate to an expression can be given any data and normally produces a value. If we for instance are describing a purely functional language, we might want to check whether the modules we include have side-effects. Therefore we would need signatures that include the types uncreative, ineffective, and stable. Modules, like *Exp*, should be fixed so that they can be reused in language descriptions, so changing the signature in *Exp* is not an option. Two solutions to the problem can be envisaged:

1. Before every type check, the user gives a signature of the function which is the target of the type checking, and the type checker infers the signatures of the other semantic functions employed in the semantic function.

2. Before every type check, the user specifies a module that contains extra type info for use in connection with type checking a particular module.

The advantage of the first suggestion is that it allows the user to see which demands it makes on the employed semantic functions when he makes demands on a semantic function. The disadvantage is that it is more difficult to implement because we have to do type inference instead of just type checking.

The second suggestion is easier to implement. The extra module given to the type checker contains more specialised versions of signatures, for instance, one could have a module that can be used to check that a module is purely functional. We have chosen this solution for our implementation.

When having more than one signature for the same semantic function that only differ with respect to the output, our type checker merges the signatures as illustrated here:

evaluate: *Exp* $\rightarrow \alpha_1$
evaluate: *Exp* $\rightarrow \alpha_2$

is merged into

evaluate: *Exp* $\rightarrow \alpha_1 \,\&\, \alpha_2$

and the equivalence (Fig. 8) is used to simplify the action type '$\alpha_1 \,\&\, \alpha_2$'.

6 Implementation

Type inference rules for copy, unfolding, and apply, involve type variables. This means that implementing the type system is more complicated than a depth-first traversal of the parse tree where the type rules are used to construct a type. The problem with rules 5, 12, and 22 is that they involve guessing types. Our implementation uses type inclusion constraints on type schemes (types with type variables). The types τ are extended with type variables θ.

To keep the implementation simple we shall use another representation of action types. The action type presented in the previous sections is readable and useful in semantic function signatures, but the following is better in an implementation, because it does not need to be normalised and type inclusion constraints with action types can more easily be simplified (see Fig. 20). We shall use the type

$$at(\tau, \tau, \tau, \tau', \tau', \tau', \tau', \tau', \tau')$$

The action type constructor at has nine arguments, one for each atomic action type, and is equivalent to the action type presented in the previous sections. The first three arguments can contain arbitrary types, and represent 'using τ', 'giving τ', and 'raising τ'. The last six arguments can contain empty, data, or a type variable θ. The arguments represent infallible, closed, terminates, uncreative, ineffective, and stable in that order. This means that an action type like 'using integer & giving boolean & closed & terminates & ineffective' is represented by the type

$$at(\mathsf{integer}, \mathsf{boolean}, \emptyset, \mathsf{data}, \emptyset, \emptyset, \mathsf{data}, \emptyset, \mathsf{data})$$

We use $\emptyset$ to indicate that the atomic action type corresponding to an argument is present and data means that it is absent. The type variables are used if the algorithm cannot determine whether an atomic action type is present or absent. The action type operator $\cup_{ac}$ produces an action type where each argument is the union of the types in the same argument in the two given action types. The type operator $\setminus$ sets the appropriate argument in an action type to data.

The idea behind the algorithm is that it transforms a set of constraints to a set of constraints in *inductive* form or an inconsistent set of constraints. Inductive sets of constraints have solutions [1]. A constraint is inductive if it has the form $\theta_j \subseteq \tau$ or $\tau \subseteq \theta_j$, and the set of variables on the top level in τ is included in $\{\theta_1, \ldots, \theta_{j-1}\}$ (see Theorem 7.2 in [1]). Here we have assumed that the type variables are numbered. The algorithm is:

1. Collect type information from the ASDF modules.
2. Traverse the parse tree of the action while generating constraints.
3. Repeat the following steps until all constraints in S are inductive and no additional inductive constraints can be added:
   - For any constraint that is not inductive, apply the lowest numbered applicable rule in Fig. 20 to simplify the set of constraints S.
   - For any pair of inductive constraints '$\tau_1 \subseteq \theta$' and '$\theta \subseteq \tau_2$' in S, add the constraint '$\tau_1 \subseteq \tau_2$' to S.
   - Stop if S is no longer consistent.
4. If we are not able to apply a type rule for each node in the parse tree, or the final set of constraints is not consistent, the action does not type check.

The constraints generated in point 2 are type inclusion constraints ($\subseteq$) and come from the use of $\leq$ in the type rules. Occurrences of the subtype relation '$\tau_1 \leq \tau_2$' (see Rules 2 and 15) are translated into '$\tau_1 \subseteq \tau_2$'.

Notice that '$\tau_1 \subseteq \tau_2$' is a constraint used in the implementation, and the interpretation is that the constraint holds if '$\tau_1 \leq \tau_2$' with the right assignment of types to the variables in τ_1 and τ_2. In the algorithm we are not going to find an assignment of types to all variables such that all constraints hold; instead we check whether an assignment exists.

The rules in Fig. 20 can be applied to the set of constraints to simplify the constraints. It is essential that the constraints on the left hand side of $\equiv$ hold for a given substitution of types to type variables if, and only if, the constraints on the right hand side do. The first rule removes the obvious constraint that does not add any extra information. Rule 2 simplifies a constraint with to product types of equal length by generating constraints comparing all of the element types. The two next rules use well known results from set theory to remove union and intersection of types that cannot be normalised. Notice that we do not have equivalent rules for intersection or

(1) $S \cup \{\emptyset \subseteq \tau\} \equiv S$

(2) $S \cup \{\tau_1 \times \ldots \times \tau_n \subseteq \tau'_1 \times \ldots \times \tau'_n\} \equiv$
$S \cup \{\tau_1 \subseteq \tau'_1\} \cup \ldots \cup \{\tau_n \subseteq \tau'_n\}$

(3) $S \cup \{\tau_1 \subseteq \tau_2 \cap \tau_3\} \equiv S \cup \{\tau_1 \subseteq \tau_2, \tau_1 \subseteq \tau_3\}$

(4) $S \cup \{\tau_1 \cup \tau_2 \subseteq \tau_3\} \equiv S \cup \{\tau_1 \subseteq \tau_3, \tau_2 \subseteq \tau_3\}$

(5) $S \cup \{\tau_1 \times \ldots \times \tau_m \oplus \theta \subseteq \tau'_1 \times \ldots \times \tau'_n\} \equiv$
$S \cup \{\tau_1 \subseteq \tau'_1, \ldots, \tau_m \subseteq \tau'_m, \theta \subseteq \tau'_{m+1} \times \ldots \times \tau'_n\}$ *when* $m \leq n$

(6) $S \cup \{\theta \oplus \tau_1 \times \ldots \times \tau_m \subseteq \tau'_1 \times \ldots \times \tau'_n\} \equiv$
$S \cup \{\theta \subseteq \tau'_1 \times \ldots \times \tau'_{n-m}, \tau_1 \subseteq \tau'_{n-m+1}, \ldots, \tau_m \subseteq \tau'_n\}$ *when* $m \leq n$

(7) $S \cup \{\tau_1 \times \ldots \times \tau_n \subseteq \tau'_1 \times \ldots \times \tau'_m \oplus \theta\} \equiv$
$S \cup \{\tau_1 \subseteq \tau'_1, \ldots, \tau_m \subseteq \tau'_m, \tau_{m+1} \times \ldots \times \tau_n \subseteq \theta\}$ *when* $m \leq n$

(8) $S \cup \{\tau_1 \times \ldots \times \tau_n \subseteq \theta \oplus \tau'_1 \times \ldots \times \tau'_m\} \equiv$
$S \cup \{\tau_1 \times \ldots \times \tau_{n-m} \subseteq \theta, \tau_{n-m+1} \subseteq \tau'_1, \ldots, \tau_n \subseteq \tau'_m\}$ *when* $m \leq n$

(9) $S \cup \{at(\tau_1, \tau_2, \ldots, \tau_9) \subseteq at(\tau'_1, \tau'_2, \ldots, \tau'_9) \equiv$
$S \cup \{\tau'_1 \subseteq \tau_1, \tau_2 \subseteq \tau'_2, \ldots, \tau_9 \subseteq \tau'_9\}$

Figure 20: Constraint simplification

union on the other side of the $\subseteq$. Such rules will not be used, as the reader can convince himself about by looking at the type rules. Union (intersection) of types only occurs as the type of output from (input to) actions, and the constraints are always generated by requiring that the output of one action be a subtype of the input given to another action (in rules 2 and 15). Rules 5 to 8 simplify constraints where application of the concatenation operator could not be normalised. In the last rule action types are removed from the set of constraints. Notice the covariance in the first argument of the action type which reflects the covariance in the atomic action type '`using` τ'

expressed in the subtype relations.

A set of constraints is inconsistent if it contains '$\tau_1 \subseteq \tau_2$' and '$\tau_1 \not\subseteq \tau_2$'. If the constraints contains '$\tau_1 \oplus \tau_2 \subseteq \tau_3$' or '$\tau_1 \subseteq \tau_2 \oplus \tau_3$' where at least two of the τ's are type variables, the simplification rules cannot simplify the constraint to a set of inductive constraints. To be on the safe side we will also consider constraint sets containing these cases to be inconsistent.

Our algorithm is almost identical to the one found in [1], so we shall not bother proving that the inductive set of constraints resulting from the simplifications has a solution if, and only if, the original set of constraints has a solution. The algorithm does not calculate a type for an action, but instead it checks that a type exist. This is sufficient because we just want to know whether an action in a semantic equation has a type and that this type is a subtype of the action type found in the signature of the corresponding semantic function.

In the Action Environment the type checker is invoked over a module, and the environment then collects type information from all imported modules before passing the semantic equations in the module together with the type information to the type checker. The result is either a message indicating that type check went well or error messages specifying where problems have been identified. The error messages can indicate where the type checker failed to apply a type rule or which action caused the constraint that made the set of constraints inconsistent. Another problem might be dead code which can occur if the left hand side of an action combinator cannot terminate in a way that allows the right hand side to be executed (for instance, if A_1 in 'A_1 catch A_2' never terminate abruptly, i.e., the type of A_1 contains '$\mathsf{raising}\ \emptyset$').

7 What can we prove?

It would be interesting to prove that the type checker can say something interesting about an action. For a normal type checker, we would want to prove that an action that type checks is well behaved. This is not possible because our type checker is liberal enough to approve actions that are not well behaved. By well behaved we mean that the action does not err because of a type error. Instead we might consider proving that if an action does not type check, then it is not well behaved, but again we run into problems. Those problems are related to the constraints that we could not simplify, and therefore resulted in an inconsistent set of constraints. The type checker rejects actions that are type correct. It appears that it is difficult to prove anything interesting about the type checker, although practical experience has shown that it is still useful.

8 Conclusion and future work

We have presented a type system for AS, which allows a soft type check of action semantic functions. The system has been implemented as a type checker operating on ASDF modules and as such it provides a useful tool when describing languages. The type checker supports the extensibility and reusability inherent in ASDF by letting the user supply the relevant type information before a type check.

Type checkers can almost always be improved to accept a bigger set of legal programs; this also holds for our semantic function type checker. With respect to the user-friendliness of the type checker, it is worth considering whether our type system can become more transparent [5, page 7]; can the user predict whether a semantic function will type check, and will he understand why it does not.

9 Acknowledgements

We would like to thank Peter D. Mosses, for initially suggesting the structure of the action types, and his many helpful comments.

We also want to thank Jan Midtgaard and Janus Dam Nielsen for proofreading earlier versions of the paper.

This work was funded by BRICS (Basic Research in Computer Science, `www.brics.dk`), which is funded by the Danish National Research Foundation.

References

[1] A. Aiken and E. L. Wimmers. Type inclusion constraints and type inference. In J. Williams, editor, *Proceedings of the Sixth Conference on Functional Programming Languages and Computer Architecture, FPCA'93*, pages 31–41. ACM Press, 1993.

[2] M. G. J. van den Brand, J. Iversen, and P. D. Mosses. An action environment. In G. Hedin and E. V. Wyk, editors, *Proceedings of the 4th Workshop on Language Descriptions, Tools and Applications, LDTA'04*, Electronic Notes in Theoretical Computer Science. Elsevier, 2004.

[3] D. Brown and D. A. Watt. JAS: A Java Action Semantics. In P. D. Mosses and D. A. Watt, editors, *Proceedings of the 2nd International Workshop on Action Semantics, AS'99*, BRICS NS-99-3, pages 43–55. Dept. of Computer Science, Univ. of Aarhus, 1999.

[4] D. F. Brown. *Sort inference in action semantics.* PhD thesis, Department of Computing Science, University of Glasgow, 1996.

[5] L. Cardelli. Type systems. In *The Computer Science and Engineering Handbook*, pages 2208–2236. CRC Press, 1997.

[6] A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping: An Algebraic Specification Approach.* AMAST Series in Computing volume 5. World Scientific, 1996.

[7] K.-G. Doh and P. D. Mosses. Composing programming languages by combining action-semantics modules. *Sci. Comput. Programming*, 47(1):3–36, 2003.

[8] K.-G. Doh and D. A. Schmidt. Extraction of strong typing laws from action semantics definitions. In B. Krieg-Brückner, editor, *Proceedings of the 4th European Symposium on Programming, ESOP'92*, LNCS volume 582, pages 151–166. Springer-Verlag, 1992.

[9] S. Even and D. A. Schmidt. Type inference for action semantics. In N. D. Jones, editor, *Proceedings of the 3rd European Symposium on Programming, ESOP'90*, LNCS volume 432, pages 118–133. Springer-Verlag, 1990.

[10] J. Iversen. Type inference for the new action notation. In P. D. Mosses, editor, *Proceedings of the 4th International Workshop on Action Semantics, AS 2002*, BRICS NS-02-8, pages 78–98. Dept. of Computer Science, Univ. of Aarhus, 2002.

[11] J. Iversen and P. D. Mosses. Constructive action semantics for Core ML. *IEE Proceedings-Software special issue on Language Definitions and Tool Generation*, 2005. to appear.

[12] S. B. Lassen, P. D. Mosses, and D. A. Watt. An introduction to AN-2, the proposed new version of Action Notation. In P. D. Mosses and H. Moura, editors, *Proceedings of the 3rd International Workshop on Action Semantics, AS 2000*, BRICS NS-00-6, pages 19–36. Dept. of Computer Science, Univ. of Aarhus, 2000. `http://www.brics.dk/~pdm/papers/LassenMossesWatt-AS-2000/`.

[13] K. D. Lee. *Action Semantics-based Compiler Generation.* PhD thesis, Department of Computer Science, University of Iowa, 1999. `http://www.cs.luther.edu/~leekent/papers/thesis.ps`.

[14] P. D. Mosses. *Action Semantics.* Cambridge Tracts in Theoretical Computer Science 26. Cambridge University Press, 1992.

[15] J. B. Wells. The essence of principal typings. In *Proceedings of the 29th International Colloquium on Automata, Languages and Programming, ICALP'02*, LNCS volume 2380, pages 913–925. Springer-Verlag, 2002. `http://www.macs.hw.ac.uk/~jbw/papers/Wells:The-Essence-of-Principal-Typ%ings:ICALP-2002.pdf`.

[16] P. Ørbæk. *Trust and Dependence Analysis.* PhD thesis, BRICS, Dept. of Computer Science, Univ. of Aarhus, Denmark, 1997.

Recent BRICS Report Series Publications

RS-04-35 **Jørgen Iversen.** ***Type Checking Semantic Functions in ASDF*****. December 2004. 29 pp.**

RS-04-34 **Anders Møller and Michael I. Schwartzbach.** ***The Design Space of Type Checkers for XML Transformation Languages*****. December 2004. 21 pp. Appears in Eiter and Libkin, editors,** ***Database Theory: 10th International Conference*****, ICDT '05 Proceedings, LNCS 3363, 2005, pages 17–36.**

RS-04-33 **Aske Simon Christensen, Christian Kirkegaard, and Anders Møller.** ***A Runtime System for XML Transformations in Java*****. December 2004. 15 pp. Appears in Bellahsene, Milo, Rys, Suciu and Unland, editors,** ***Database and XML Technologies: Second International XML Database Symposium*****, XSym '04 Proceedings, LNCS 3186, 2004, pages 143–157. Supersedes the earlier BRICS report RS-03-29.**

RS-04-32 **Philipp Gerhardy.** ***A Quantitative Version of Kirk's Fixed Point Theorem for Asymptotic Contractions*****. December 2004. 9 pp.**

RS-04-31 **Philipp Gerhardy and Ulrich Kohlenbach.** ***Strongly Uniform Bounds from Semi-Constructive Proofs*****. December 2004. 31 pp.**

RS-04-30 **Olivier Danvy.** ***From Reduction-Based to Reduction-Free Normalization*****. December 2004. 27 pp. Invited talk at the** ***4th International Workshop on Reduction Strategies in Rewriting and Programming*****, WRS 2004 (Aachen, Germany, June 2, 2004). To appear in ENTCS.**

RS-04-29 **Małgorzata Biernacka, Dariusz Biernacki, and Olivier Danvy.** ***An Operational Foundation for Delimited Continuations in the CPS Hierarchy*****. December 2004. iii+45 pp. Appears in Thielecke, editor,** ***4th ACM SIGPLAN Workshop on Continuations*****, CW '04 Proceedings, Association for Computing Machinery (ACM) SIGPLAN Technical Reports CSR-04-1, 2004, pages 25–33.**

RS-04-28 **Mads Sig Ager, Olivier Danvy, and Jan Midtgaard.** ***A Functional Correspondence between Monadic Evaluators and Abstract Machines for Languages with Computational Effects*****. December 2004. 44 pp. Extended version of an article to appear in** ***Theoretical Computer Science*****.**