A Type-Based Approach to Separating Protocol from Application Logic
A Case Study in Hybrid Computer Programming

Geoffrey C. Hulette[1], Matthew J. Sottile[2], and Allen D. Malony[1]

[1] University of Oregon, Eugene, OR
[2] Galois, Inc., Portland, OR

Abstract. Numerous programming models have been introduced to allow programmers to utilize new accelerator-based architectures. While OpenCL and CUDA provide low-level access to accelerator programming, the task cries out for a higher-level abstraction. Of the higher-level programming models which have emerged, few are intended to co-exist with mainstream, general-purpose languages while supporting tunability, composability, and transparency of implementation. In this paper, we propose extensions to the type systems (implementable as syntactically neutral annotations) of traditional, general-purpose languages can be made which allow programmers to work at a higher level of abstraction with respect to memory, deferring much of the tedium of data management and movement code to an automatic code generation tool. Furthermore, our technique, based on formal term rewriting, allows for user-defined reduction rules to optimize low-level operations and exploit domain- and/or application-specific knowledge.

1 Introduction

Programming for hybrid architectures is a challenging task, in large part due to the partitioned memory model they impose on programmers. Unlike a basic SMP, devices must be set up and torn down, processing synchronized, and data explicitly allocated on a particular device and moved around within the memory hierarchy. Programming systems such as CUDA[1] and OpenCL[2] provide an interface for these operations, but they are quite low-level. In particular, they do not distinguish between the high-level computational and application logic of a program, and the *protocol logic* related to managing heterogeneous devices. As a result, the different types of program logic invariably become entangled, leading to excessively complex software that is prohibitively difficult to develop, maintain, and compose with other software. The problem we have described is pervasive in programming for hybrid architectures; in this paper, we will focus on the specific instance of this problem presented by GPU-based accelerators.

We present a high-level programming language called *Twig*, designed for expressing protocol logic and separating it from computational and application logic. Twig also supports automated reasoning about composite programs that

C. Kaklamanis et al. (Eds.): Euro-Par 2012, LNCS 7484, pp. 40–51, 2012.
© Springer-Verlag Berlin Heidelberg 2012

can, in many cases, avoid problems such as redundant memory copying. This allows Twig programs often to retain the high performance of a lower-level programming approach.

Crucially, Twig's role in the programming toolchain is to generate code in a mainstream language, such as C. The generated code is easily incorporated into the main program, which is then compiled as usual. This minimizes the complexity that Twig adds to the build process, and allows Twig code to interact easily with existing code and libraries.

Twig achieves these goals by using data types to direct the generation of code in the target language. In particular, we augment existing data types in the target language with a notion of *location*, e.g., an array of floats located on a GPU, or an integer located in main memory. In the following sections, we first present related work, and then describe Twig's code generation strategy and core semantics. Finally, we present an example demonstrating the use of located types to generate code for a GPU. In the example, we also show how Twig programs can be automatically rewritten in order to minimize data movement.

2 Related Work

Twig was inspired in part by Fig[3]. In that project, a similar formal approach was used to express bindings between different programming languages. In our experience, multi-language programming has much in common with programming hybrid systems. The overlaps include memory ownership and management, data marshaling, and managing the flow of program control across the language or device boundary. Our work builds upon the approach in Fig, and in particular aims to provide a general-purpose tool not tied to the Moby[4] programming language.

Numerous systems have been created in recent years that provide an abstraction above low-level interfaces such as OpenCL or CUDA. These include the PGI Accelerate model[5], the HMPP programming system[6], and Offload[7]. Interestingly, Offload, like Twig, uses locations encoded in the type system. While these systems provide an effective high-level abstraction, they offer little room for tuning the low-level interface to the accelerator. Twig provides a simple method for user-definable rewriting of programs, which allows architecture-, domain-, and even application-specific optimizations to be realized.

Furthermore, in large applications it is infeasible to assume that all developers of the various components will use the same high-level abstraction. This makes program composition challenging, since it may be unclear how the objects generated by independent programming systems interacts. Twig adopts a code generation approach in which a single, low-level target (such as CUDA) is used. This approach solves the composability problem, since all Twig code maps to a single "lingua franca" for programming the hybrid system.

Sequoia[8] is a language and runtime system for programming hybrid computers. It allows programmers to explicitly manage the memory hierarchy, while retaining program portability across architectures. Although Sequoia is based

on C++, it is intended as a complete programming environment, not as a way to extend existing programs with hybrid computation.

Code generation approaches have had notable success in the computational science field, an exemplar being the Tensor Contraction Engine (TCE)[9]. The TCE allows computational chemists to write tensor contraction operations in a high level language, and then generates the corresponding collections of loops that implement the operations. Unlike Twig, the TCE is quite specialized, being of use only to programmers working with tensor-based computations.

3 Method Overview

In Twig, we write *rules* which express some high-level operation, such as kernel execution or copying data to a device, as a function on *types*. A Twig program is evaluated with a type given as input. The output is another type, transformed by the combined rules of the program. As a side-effect, C code is generated which performs the transformation on values in C. This basic idea is formalized in Sec. 5.

Types in Twig are based on the set provided by C, but may be augmented with additional information. For GPU programming, we augment standard data types with a *location*. The location information describes where the data is stored in memory; in this case, either in the main system memory or on the GPU. For example, we can represent an array of `int`s on the CPU with the Twig type `array(int)`. The same type located on the GPU is `gpu(array(int))`. Any standard type may be wrapped inside the `gpu` type constructor.

Note that location information is only used by Twig during evaluation and code generation. In particular, it may not be reflected in the types for the generated code. If we are generating CUDA code, for example, the generated type for both `gpu(array(int))` and `array(int)` is simply a C pointer to `int` (i.e., `int *`). In this case, the location information is erased during the code generation phase. For other target languages or APIs that have a notion of location, the information could be preserved in the target data types.

By augmenting basic data types with location information, we ensure that rules must be specific to the GPU in order to operate on GPU data. For example, a rule

```
[gpu(array(float)) -> gpu(array(int))]
```

converts an “array of floats” data type to an “array of integers” type if and only if the type describes data located on the GPU. If the type describes data located elsewhere, its type must be “converted” (i.e., the data copied to the device) with a rule such as

```
[array(float) -> gpu(array(float))]
```

This simple scheme enables Twig to reason about requirements for data motion.

It is important to understand that rules such as those given above describe transformations on *data types*, not on the data themselves. It falls to the code that is generated as a consequence of successful application of these rules to perform the promised conversion on the actual data. Code generation is described in Sec. 4.

Our scheme could be extended to support multiple GPU devices, with each device corresponding to a unique located type. In fact, we think that located types could be useful in a variety of situations; this is a topic of ongoing work.

4 Code Generation

To generate code, Twig uses an abstract, language-independent system with a small number of basic operations. The relative simplicity of the model is motivated by Twig's semantics, described in Sec. 5. It is helpful in clarifying the precise operations which Twig supports, without getting bogged down in the (potentially quite complicated) details of outputting code for a particular language.

Twig generates code in units called *blocks*. A block of code represents anything that performs some operation on a set of inputs in order to produce a set of outputs. Blocks have zero or more inputs and/or outputs. Blocks can be combined in two different ways: *sequentially*, or *in parallel*. These operations are described below.

Our current implementation of this model generates C code, although the model is general enough to generate other languages as well. Figures 1(a) and 1(b) each depict a different basic block that generates C.

4.1 Block Composition

As mentioned above, Twig provides two fundamental operations on blocks. The first is *sequential* composition, which we represent formally as addition ($+$) on blocks. Sequencing connects two blocks of code by "wiring" the outputs of the first block into the inputs of the second (see Fig. 1(c)). In C, this is done by creating uniquely-named temporary variables which are substituted into the original blocks.

The second operation is *parallel* composition, where two blocks are combined so as to execute independently of one another, but to appear as one single block (see Fig. 1(d)). We represent this operation as multiplication ($\times$).

4.2 Identity Blocks

Twig's formal semantics require the definition of a set of special *identity* blocks. An identity block I_n has n inputs and n outputs ($n > 0$). Its function is, as its name implies, to simply pass each of its inputs through, unchanged, to the corresponding output. In Twig's semantics we use I_n as a kind of "no-op." We also use I in place of I_n when the value of n is implied from the context.

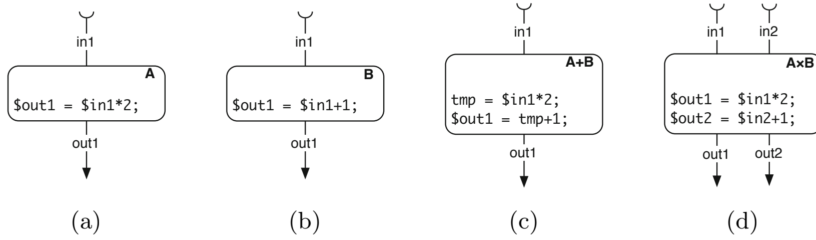

Fig. 1. Code generation using blocks. A (a) and B (b) are basic blocks. $A + B$ (c) is the sequential composition of A and B. $A \times B$ (d) is the parallel composition of A and B.

Identity blocks are subject to a few rules, which we describe here informally and only briefly. First, I_n are left- and right-identity elements under sequential composition, i.e., $I + x = x + I = x$ for all blocks x. Second, we can define parallel composition of identity blocks is defined by summing their size, i.e., $I_n \times I_m = I_{n+m}$.

5 Twig's Semantics

Twig is based on a core semantics for term rewriting called System S[10], augmented with code generation and specialized to operate on types instead of general terms. Twig uses the operators in System S to combine primitive *rules* into more complex transformations on types. These transformations are then applied to a given type, resulting in a new type and potentially generating code as a side effect. In this section we describe the Twig language.

We present an abbreviated and relatively informal description of Twig's semantics, focusing on the features used to support GPU programming. The full semantics will be described in a forthcoming paper.

Twig does not currently provide any built-in constructs for expressing general recursive expressions, including loops. We are working to address this limitation in our current work. At the moment, Twig can generate loops contained within a single generated block and, of course, Twig can also be used to generate a block of code for inclusion within a loop body.

5.1 Values

Values in Twig can be any valid *term* representing a type in the target language. Terms are tree structured data with labeled internal nodes. Examples of terms include simple values like `int` and `float`, as well as compound types like `ptr(int)`, which might represent a pointer to an integer in C.

The mapping between terms in Twig and types in the target language is a configuration option. Furthermore, the mapping need not be injective, i.e. users are free to have multiple values in Twig map to the same type in C. For example, you might use distinct Twig values `string` and `ptr(char)`, but map both to a pointer to `char` (`char *`) in C.

Twig also includes support for terms representing groups of values, i.e. *tuples*, and operations on groups. We omit these semantics here for lack of space.

5.2 Rules

The fundamental components of a Twig program are called *primitive rules*. A primitive rule describes a transformation from one term to another. For example, in C it is easy to convert an integer value to floating point, and we can write this rule in Twig as follows:

```
[int -> float]
```

The term to the left of the arrow is the input, and the term to the right is the output. In this example, the rule says that if and only if the input to the rule is the term `int`, then the output will be the term `float`. If the input is not `int`, then the output will be the special value $\perp$, which can be read as "undefined" or "failure."

Primitive rules will typically generate code as a side effect of successful application. To associate a block of code with a rule, the programmer puts it immediately after the rule definition and surrounds it with braces. The brace symbols are configurable; here we use `<|` and `|>`. For example:

```
[int -> float] <| $out = (float)$in; |>
```

When the code is generated, Twig will create temporary variables for `$in` and `$out` and ensure that the various bookkeeping details, such as variable declarations, are handled. If there are multiple inputs or outputs, then the relevant placeholders are enumerated; e.g., `$in1`, `$in2`, and so on.

Note that Twig does not check the generated code for correctness – the generation procedure is essentially syntactic. This approach is similar to the strategy used tools such as SWIG[11].

5.3 Formal Semantics

In the following formal semantics, let t range over terms, m over code block expressions, and s over rule expressions, i.e., a primitive rule or a sub-expression built with operators.

Primitive Rules. A primitive rule s transforms a term t to another term t' with generated code m:

$$t \xrightarrow{s} (t', m)$$

if the application of rule s to value t succeeds. If no code is given for the rule, then m is the identity element, I (see Sec. 4). If the application of s to t fails, e.g., if t does not match the pattern in s, then

$$t \xrightarrow{s} \perp$$

Note that a code block is not emitted in this case.

Operators. Rules can be combined into more complex expressions using operators. The most useful of these is the *sequence* operator (note the distinction from the + operator for code blocks described in Sec. 4). A sequence chains the application of two rules together, providing the output of the first to the input of the second, and failing if either rule fails (see Fig. 2). With this operator, simple rules can be composed into multi-step transformations.

$$\frac{t \xrightarrow{s_1} (t', m_1) \quad t' \xrightarrow{s_2} (t'', m_2)}{t \xrightarrow{s_1;s_2} (t'', m_1 + m_2)} \qquad \frac{t \xrightarrow{s_1} \bot}{t \xrightarrow{s_1;s_2} \bot} \qquad \frac{t \xrightarrow{s_1} (t', m) \quad t' \xrightarrow{s_2} \bot}{t \xrightarrow{s_1;s_2} \bot}$$

Fig. 2. Semantics for sequence operator

Another important operator is *left-biased choice*. Choice expressions will try the first rule expression, and if it succeeds then its output is the result (see Fig. 3) of the expression. If the first rule fails (i.e., results in $\bot$), then the second rule is tried. This operator allows different paths to be taken, and different code to be generated, depending on the input type.

$$\frac{t \xrightarrow{s_1} (t', m)}{t \xrightarrow{s_1|s_2} (t', m)} \qquad \frac{t \xrightarrow{s_1} \bot \quad t \xrightarrow{s_2} (t', m)}{t \xrightarrow{s_1|s_2} (t', m)} \qquad \frac{t \xrightarrow{s_1} \bot \quad t \xrightarrow{s_2} \bot}{t \xrightarrow{s_1|s_2} \bot}$$

Fig. 3. Semantics for left-biased choice

Fig. 4 gives the formal semantics for some of Twig's other basic operators. These include constant operators and operators which discard their results.

$$\frac{}{t \xrightarrow{\texttt{id}} (t, I)} \qquad \frac{}{t \xrightarrow{\texttt{fail}} \bot} \qquad \frac{t \xrightarrow{s} (t', m)}{t \xrightarrow{?s} (t, I)} \qquad \frac{t \xrightarrow{s} \bot}{t \xrightarrow{?s} \bot} \qquad \frac{t \xrightarrow{s} (t', m)}{t \xrightarrow{\neg s} \bot} \qquad \frac{t \xrightarrow{s} \bot}{t \xrightarrow{\neg s} (t, I)}$$

Fig. 4. Semantics for basic operators

Twig also provides some special operators for tuples. These are not needed for this paper, so we omit further discussion.

Named Expressions. Twig allows rules and rule expressions to be assigned to names. The name can be used in place of the rule itself within expressions. For example:

```
intToFloat = [int -> float] { ... }
```

A Twig program is a list of such name/expression assignments. There is a special name, `main`, which designates the top-level expression for the program.

5.4 Reductions

Reductions are a mechanism provided within Twig as a way to automatically simplify expressions. Reductions can be used to exploit application or domain knowledge about primitive rules, and as such are usually developed alongside a set of rules.

As an example, consider the following two rules:

```
intToFloat = [int -> float] {
  $out = (float)$in;
}

floatToInt = [float -> int] {
  $out = (int)$in;
}
```

and the expression

```
intToFloat;floatToInt
```

We would most likely consider this conversion to be redundant and we should eliminate it wherever possible. We can tell Twig to do this with the following reduction rule:

```
reduce intToFloat;floatToInt => id
```

This statement instructs Twig to replace any subexpression `intToFloat; floatToInt` with the identity rule, `id`, anywhere it occurs within the program. Recall that `id` is the identity rule; it simply passes the value through unchanged.

Twig comes equipped with some standard reductions by default. These reductions rely on the meaning of Twig's combinators to normalize expressions. For example, we can replace subexpressions of the form `id;X` with `X`, where `X` represents any subexpression.

Twig's reductions are based on the theory of term rewriting; for a formal discussion see [12]. In this case, Twig's expressions constitute the terms. There are some subtleties with reductions, e.g., they must be developed carefully to avoid circular reductions.

6 Implementation

Our implementation of Twig is written in Haskell. Twig expects as input a `.twig` file containing a list of named rule expressions along with a `main` rule expression, as described in Sec. 5.3. It also expects an initial value (i.e. a term, representing a C type), which will be used as the input to the main rule expression. Twig must also be configured with a mapping from terms to C types. Currently, this mapping is provided with a simple key/value text file.

Generated code may optionally be wrapped in a C function body, with parameters corresponding to the inputs, and return value corresponding to the output. The generated code may be redirected to a separate file and included in a C program using an `#include` directive.

7 Example

Now we present an example program written in Twig. The code in Fig. 5 demonstrates how Twig is used, and how reductions can eliminate redundant memory copies.

```
copyToGPU=[array(float) -> gpu(array(float))] <|
  cudaMalloc((void **)&$out,SIZE);
  cudaMemcpy($out,$in,SIZE,cudaMemcpyHostToDevice);
|>
copyFromGPU=[gpu(array(float)) -> array(float)] <|
  $out = malloc(SIZE * sizeof(float));
  cudaMemcpy($out,$in,SIZE,cudaMemcpyDeviceToHost);
|>
kernel(k)=[gpu(array(float)) -> gpu(array(float))] <|
  $k <<<N_BLOCKS,BLOCK_SIZE>>>($in, N);
  $out = $in;
|>
runKernel(k)=copyToGPU;kernel(k);copyFromGPU
main=runKernel(<|foo|>);runKernel(<|bar|>)
reduce copyFromGPU;copyToGPU => id
```

Fig. 5. Twig code example

This example is quite simple, in the interest of brevity and clarity. We omit setup and teardown logic, and assume that the array size, block size, and other parameters are simple constants. A real application would probably pass these values around using more complex rules.

The first three rules definitions are primitives for moving data to and from the GPU (`copyToGPU` and `copyFromGPU`), and for invoking a kernel transformation on the array in GPU memory (`kernel`). The rules `kernel` and `runKernel` are parameterized by `k`, whose value is inserted directly into the generated code.

The `runKernel` rule will perform a single logical "function" on the GPU. Note that this rule will be semantically valid in any context where it appears, since it ensures that the data is first moved onto the GPU, the kernel is executed, and then the data is copied back. To the programmer, `runKernel` appears to perform a function on a local array – a considerably simpler than the abstraction presented by OpenCL or CUDA.

The `main` rule is the top level of the program. This example executes two kernels in sequence with two invocations of `runKernel`. As noted above, by design each invocation would normally result in a copy to and from the GPU – a conservative strategy. Since the data is not modified in between GPU calls, on its own this expression would generate a redundant copy in between the calls to `foo` and `bar`. To see why, we can trace the execution of the Twig program. First, variable names are substituted with the expressions they denote, so `main` goes from:

```
main = runKernel(foo);runKernel(bar)
```

to

```
main = copyToGPU;kernel{foo};copyFromGPU;
       copyToGPU;kernel{bar};copyFromGPU
```

Evaluating this expression on the type `array(float)` will generate the following code.

```
float *tmp01,*tmp02,*tmp03,*tmp04,*tmp05,*tmp06,*tmp07;
cudaMalloc((void **)&tmp02,SIZE);
cudaMemcpy(tmp02,tmp01,SIZE,cudaMemcpyHostToDevice);
foo <<<N_BLOCKS,BLOCK_SIZE>>> (tmp02, N);
tmp03 = tmp02;
tmp04 = malloc(SIZE * sizeof(float));
cudaMemcpy(tmp04,tmp03,SIZE,cudaMemcpyDeviceToHost);
cudaMalloc((void **)tmp05,SIZE);
cudaMemcpy(tmp05,tmp04,SIZE,cudaMemcpyHostToDevice);
bar <<<N_BLOCKS,BLOCK_SIZE>>> (tmp05,N);
tmp06 = tmp05;
tmp07 = malloc(SIZE * sizeof(float));
cudaMemcpy(tmp07,tmp06,SIZE,cudaMemcpyDeviceToHost);
```

Notice that this code, while correct, contains a redundant copy! The problem arises because we sequenced the two kernel operations, which introduces the sub-expression `copyFromGPU;copyToGPU`. This sub-expression will copy data from the GPU to main memory, and then immediately back to the device. We solve this problem using a *reduction*, as described in Sec. 5.4. The line

```
reduce copyFromGPU;copyToGPU => id
```

instructs Twig to search for the expression `copyFromGPU;copyToGPU` and replace it with `id`, the identity transformation. After the reduction step, the expanded version of `main` has the extra copies removed:

```
main = copyToGPU;kernel{foo};id;kernel{bar};copyFromGPU
```

In fact, Twig's built-in reduction rules would remove the spurious `id` as well, although this has no bearing on the meaning. Now Twig will generate this code:

```
float *tmp01,*tmp02,*tmp03,*tmp04,*tmp05;
cudaMalloc((void **)&tmp02,SIZE);
cudaMemcpy(tmp02,tmp01,SIZE,cudaMemcpyHostToDevice);
foo <<<N_BLOCKS,BLOCK_SIZE>>> (tmp02, N);
tmp03 = tmp02;
bar <<<N_BLOCKS,BLOCK_SIZE>>> (tmp03,N);
tmp04 = tmp03;
tmp05 = malloc(SIZE * sizeof(float));
cudaMemcpy(tmp05,tmp04,SIZE,cudaMemcpyDeviceToHost);
```

This code does not contain the extraneous copying. Although this example is simple, it demonstrates the power of reductions. The reduction rule given here would probably be paired with the `copyToGPU` and `copyFromGPU` rules in a module intended for consumption by domain programmers, allowing them to perform GPU operations without worrying about the design of the rules. Sophisticated users, however, could add their own rules or even application-specific reductions, enabling very powerful and customizable code generation based on domain-specific logic.

8 Future Work

We are working on expanding the Twig language with a notion of *functors*. Functors cleanly capture most cases in which users might need to generate loop constructs, allocate/free patterns, or other protocols that require a notion of *context*.

We are also investigating a number of ways in which Twig might be more closely integrated with mainstream coding practices. For example, we imagine that it may be possible for Twig code to live in the background, and express protocol logic through declarative annotations in the application code.

9 Conclusion

We have introduced the concept of separating the protocol logic inherent to hybrid systems from the computational and application logic of a program. We have demonstrated that a type-based approach can enforce this separation by making explicit in data types information related to both data location, and the representation of the data itself. By doing so, we allow the protocol logic of a program to be expressed via operations exclusively on located types. Many explicit programming chores become implicit features of the generated code, such as declaring intermediate values or reducing redundant memory movement. Finally, by adopting a code generation approach, we show that users of these higher level abstractions are not prohibited from both tuning the resultant code and composing independently developed programs that utilize standardized hybrid programming libraries like OpenCL or CUDA.

Acknowledgements. This work was supported in part by the Department of Energy Office of Science, Advanced Scientific Computing Research.

References

1. Sanders, J., Kandrot, E.: CUDA By Example: An Introduction To General-Purpose GPU Programming (July 2010)
2. Khronos OpenCL Working Group: The OpenCL Specification Version 1.0
3. Reppy, J., Song, C.: Application-specific foreign-interface generation. In: GPCE 2006, pp. 49–58 (October 2006)

4. Fisher, K., Reppy, J.: The design of a class mechanism for Moby. In: SIGPLAN 1999, pp. 37–49 (May 1999)
5. Wolfe, M.: Implementing the PGI accelerator model. In: GPGPU 2010 (2010)
6. Dolbeau, R., Bihan, S., Bodin, F.: HMPP: A hybrid multi-core parallel programming environment. In: GPGPU 2007 (2007)
7. Cooper, P., Dolinsky, U., Donaldson, A.F., Richards, A., Riley, C., Russell, G.: Offload – Automating Code Migration to Heterogeneous Multicore Systems. In: Patt, Y.N., Foglia, P., Duesterwald, E., Faraboschi, P., Martorell, X. (eds.) HiPEAC 2010. LNCS, vol. 5952, pp. 337–352. Springer, Heidelberg (2010)
8. Fatahalian, K., Knight, T., Houston, M., Erez, M., Horn, D., Leem, L., Park, H., Ren, M., Aiken, A., Dally, W., Hanrahan, P.: Sequoia: Programming the memory hierarchy. In: SC 2006 (November 2006)
9. Baumgartner, G., et al.: Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models. Proceedings of the IEEE (2005)
10. Visser, E., el Abidine Benaissa, Z.: A core language for rewriting. Electronic Notes in Theoretical Computer Science 15, 422–441 (1998)
11. Beazley, D.M.: Automated scientific software scripting with SWIG. Future Generation Computer Systems 19, 599–609 (2003)
12. Baader, F., Nipkow, T.: Term rewriting and all that. Cambridge University Press, New York (1998)